# So Above

**Category**: “The Magician”

## Description

The challenge is provided with a Docker container, which entrypoint is the Ruby intepreter, called on the driver.rb script.

This script does a series of operation n times, where n is a value specified in the environment variable NUMBER\_TO\_GENERATE. For each iteration, the following operations are done.

First, the script generates and saves in /chall/challs/generated.c the source code of a C function, using a template it embeds. The function looks like

bool quick\_maths(double run) {  
 // start of a bunch of operations of type  
 run = run operator value;  
 ...  
 run = run operator value;  
 // end of operations  
 return (run == final\_value);  
}

Each (value, operator) pair and the final\_value are randomly generated by the script while creating the source code. The allowed operators are addition, division, and subtraction, while a value is between 0 and UINT32\_MAX.

When the generation of generated.c is completed, the script compiles the main.c file into the /chall/challs/generated binary. The main function of the binary - reads from stdin a double into the candidate variable - calls quick\_maths(candidate). - if quick\_maths returns 0, it prints cool :), otherwise it prints hmm… not happy :(.

Finally, the script invokes the TypeScript Deno interpreter with a script we provide on the challenge’s frontend, and uses its output as the input to run the previously compiled binary.

If the compiled binary prints cool :) for each iteration, in the end, the script prints the flag.

## Solution

The goal of the challenge is to write a TypeScript script which prints on stdout the numeric value for which the quick\_maths function returns 1 so that the main prints cool :).

Unfortunately, we cannot read the source code of quick\_maths, which is in the /chall/challs/generated.c file. The CLI option --allow-read=/chall/challs/generated limits the files accessible from the Deno interpreter. We can read from Deno only the generated X86\_64 binary executable. So, we need to recover the (operand, value) pair of each operation by parsing the executable.

Fortunately, from the TypeScript template provided with the challenge, we notice that it is possible to use the iced-x86 library, which is an x86 disassembler for TypeScript.

By compiling several times the source code with different sets of operations, we notice that the compiler always places the quick\_maths function at an offset of 0x11C9 from the beginning of the executable. By disassembling the binary with IDA Freeware, we notice that the quick\_maths function has the following structure

* Function prologue

.text:0x11C9 push rbp  
.text:0x11CA mov rbp, rsp

* This group of instructions, repeated for each operation in the source code

.text:0x11CD movsd qword ptr [rbp-8], xmm0  
.text:0x11D2 movsd xmm1, qword ptr [rbp-8]  
.text:0x11D7 movsd xmm0, cs:qword\_2040  
.text:0x11DF addsd xmm0, xmm1

There the most interesting instructions are the last two. At 0x11D7 the operation’s value is loaded from the .rodata section, and the instruction at 0x11DF computes the operation.

* The function ending

.text:0x1317 movsd qword ptr [rbp-8], xmm0  
.text:0x131C movsd xmm0, qword ptr [rbp-8]  
.text:0x1321 ucomisd xmm0, cs:qword\_20B8  
.text:0x1329 setnp al  
.text:0x132C mov edx, 0  
.text:0x1331 movsd xmm0, qword ptr [rbp-8]  
.text:0x1336 ucomisd xmm0, cs:qword\_20B8  
.text:0x133E cmovnz eax, edx  
.text:0x1341 pop rbp  
.text:0x1342 retn

There the value of run is compared with final\_value.

We recover the correct input value by reading the binary executable with Deno.readFile('/chall/challs/generated') API call and getting the pair (operator, value) for each operation.

Let’s call base the address of the currently processed operation, that is, the address of the last instruction in the group of instructions of an operation. For each operation to recover, we do the following operations:

* Read the byte at base + 2 to get the operand of the current operation. In fact, addsd, subsd, and divsd instructions are encoded with the same bytes but the third, which represents the operation type. 0x5c correspond to subsd, 0x58 to addsd and 0x5e to divsd.
* Read the bytes of movsd instruction at base - 8 and disassemble them, using iced-x86, to recover the address where the current value is stored.
* Read 8 bytes at the address recovered in the previous step to get the value of the operation. Since the values are stored as double, we convert them to an integer value.
* Increment base of 22 to go at the next operation (the group of instructions of a single operation is long 22 bytes)

We know that we are done with retrieving the operations when at base + 2 there is 0xc0, the last byte of the setnp al instruction. Since at base - 8 there is the ucomisd xmm0, cs:qword\_20B8, we read its bytes and disassemble it to get the address where final\_value is stored. We recover final\_value as we do for the other values.

In the end, we invert the operations and get the input value for which quick\_maths returns 1.

## Solution script

// 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"  
  
let debug: (\_jawn: any) => void  
  
if (await Deno.permissions.query({name: "env", variable: "SOLVER\_DEBUG"}) &&  
 Deno.env.get('SOLVER\_DEBUG')) {  
 const enc = new TextEncoder()  
  
 debug = (jawn: any) => {  
 Deno.stderr.writeSync(enc.encode(Deno.inspect(jawn) + "\n"))  
 }  
} else {  
 debug = (\_jawn: any) => {}  
}  
  
let file\_data = await Deno.readFile('/chall/challs/generated')  
let base = 0x11DF  
  
function getOperandAddress(base) {  
 const exampleBitness = 64;  
 base -= 8  
 const exampleCode = file\_data.slice(base, base + 8)  
  
 const decoder = new iced.Decoder(exampleBitness, exampleCode, iced.DecoderOptions.None)  
 decoder.ip = BigInt(base)  
  
 const instructions = decoder.decodeAll()  
 const formatter = new iced.Formatter(iced.FormatterSyntax.Nasm);  
  
 // Change some options, there are many more  
 formatter.digitSeparator = "`";  
 formatter.firstOperandCharIndex = 10;  
  
 let s = formatter.format(instructions[0]).split(' ').reverse()[0].replace('h]', '')  
 return parseInt(s, 16)  
}  
  
function bytesToDouble(data: number[]) {  
 var buf = new ArrayBuffer(8);  
 var view = new DataView(buf);  
  
 data.reverse().forEach(function (b, i) {  
 view.setUint8(i, b);  
 });  
  
 return view.getFloat64(0);  
}  
  
function getOperand(address) {  
 let byte\_aaa = [0,0,0,0,0,0,0,0]  
 for (let j = 0; j < 8; j++) {  
 byte\_aaa[j] = file\_data[address + j]  
 }  
  
 return bytesToDouble(byte\_aaa)  
}  
  
const operations: [string, number][] = [];  
  
while (true) {  
 let curr\_operation = file\_data[base + 2]  
  
 if (curr\_operation == 0xc0)  
 break  
  
 else {  
 let operand\_address = getOperandAddress(base)  
 let operand = getOperand(operand\_address)  
  
 if (curr\_operation == 0x5c) {  
 operations.push(['-', operand])  
 }  
  
 if (curr\_operation == 0x58) {  
 operations.push(['+', operand])  
 }  
  
 if (curr\_operation == 0x5e) {  
 operations.push(['/', operand])  
 }  
 }  
  
 base += 22  
}  
  
let result = getOperand(getOperandAddress(base))  
  
for (const [op, value] of operations.reverse()) {  
 if (op == '+') {  
 result -= value  
 }  
 if (op == '-') {  
 result += value  
 }  
 if (op == '/') {  
 result \*= value  
 }  
 }  
  
console.log(result)