ECSC Estonia Prequalifier - Stern Broccoli

Upon opening the binary in Binary Ninja and taking a look at the main function from afar, it looks like the user input is being modified somehow and then compared to the one stored in the binary:

```
00001330  int32_t main(int32_t argc, char** argv, char** envp)

00001330  {
0000133f      void* fsbase;
0000133f      int64_t rax = *(uint64_t*)((char*)fsbase + 0x28);
00001352      char input_buffer[0x40];
00001352      __builtin_memset(&input_buffer, 0, 0x40);
000013c9      double changeable_buffer[0x40];
000013c9      __printf_chk(1, "Please tell me the code: ", envp, __builtin_memset(&changeable_buffer, 0, 0x200));
000013e2      fgets(&input_buffer, 0x40, stdin);
000013e7      int16_t or_value = -0x100;
000013e7
0000143f      for (int64_t i = 0; i != 64; )
0000143f      {
00001404          double obfuscated_double = obfuscate((((int16_t)input_buffer[i]) | or_value));
00001409          int32_t usually_i = ((int32_t)(i + 7));
00001409
0000140e          if (i >= 0)
0000140e              usually_i = i;
0000140e
0000141b          uint32_t always_zero_value = ((i >> 31) >> 29);
0000142d          // swaps first 3 bits and last 3 bits
0000142d          changeable_buffer[(((int64_t)(usually_i >> 3)) + (((int64_t)(((always_zero_value + i) & 7) - always_zero_
00001432          i += 1;
00001436          or_value -= 0x100;
0000143f      }
0000143f
00001452      if (check_validity(&changeable_buffer, &data_4020) == 0)
0000148b          puts("Sorry, try again!");
00001452      else
0000145b          puts("Congratulations, you found the f…");
0000145b
00001468      *(uint64_t*)((char*)fsbase + 0x28);
```

(The variable names and function names are already renamed here since I am writing this writeup after solving the challenge, originally they would be more random).

With these types of challenges, it's usually best to start reversing from the end, i.e from the function that verifies the user input.

In the `check_validity` (name given by me) function, we can see that it consists of two parts:

```
0000129c     do
0000129c     {
0000124f         double* flag_buf_offset = flag_buffer;
00001252         double* i_1 = current_s_block_start_addr;
00001252
0000128b         // loops a total of 8 times over the flag buffer, each time
0000128b         // starting from an offset moved forward by 1
0000128b         do
0000128b         {
00001258             double_1 = *(uint64_t*)i_1;  // this is zero
0000125c             double* j_1 = flag_buf_offset;
0000125f             uint64_t* inpt_buffer_modified_2 = inpt_buffer_modified_1;
0000125f
00001279             do
00001279             {
00001266                 _128bit_double = (*(uint64_t*)inpt_buffer_modified_2 * *(uint64_t*)j_1);
0000126a                 double_1 = (double_1 + _128bit_double);
0000126e                 // Move forward to the next element
0000126e                 inpt_buffer_modified_2 = &inpt_buffer_modified_2[1];
00001272                 j_1 = &j_1[8];
00001279             } while (inpt_buffer_modified_2 != inpt_buffer_modified_cur_block_end);
00001279
0000127b             *(uint64_t*)i_1 = double_1;  // fill buffer
00001280             i_1 = &i_1[1];
00001284             flag_buf_offset = &flag_buf_offset[1];
0000128b         // loops a total of 8 times over the flag buffer, each time
0000128b         // starting from an offset moved forward by 1
0000128b         } while (flag_buf_offset != &flag_buffer[8]);
0000128b
0000128d         inpt_buffer_modified_1 = &inpt_buffer_modified_1[8];
00001291         // Look at the next block of 8 values
00001291         inpt_buffer_modified_cur_block_end = &inpt_buffer_modified_cur_block_end[8];
00001295         current_s_block_start_addr = &current_s_block_start_addr[8];
0000129c     // fills buffer s 8 bytes at a time, the resulting buffer is the
0000129c     // flag buffer * input buffer (elements at same index are
0000129c     // multiplied and set as the value of s)
0000129c     } while (current_s_block_start_addr != &s_end_address);
```

And

```
000012f9        while (true)
000012f9        {
000012f9            int64_t rax_1 = 0;
000012f9
000012df            while (true)
000012df            {
000012df                double_1 = 1.0;
000012df
000012e5                // The first loop double_1 will be 1 for the first number
000012e5                // out of the 8, for the second one the second etc.
000012e5                if (rdx_1 != rax_1)
000012e7                    double_1 = ((int64_t){0});
000012e7
000012c7                _128bit_double = (cur_s_buffer_start[rax_1] - double_1);
000012d3                int64_t result;  // gets absolute value
000012d3
000012d3                if (_mm_and_pd(_128bit_double, ((uint128_t)0x7fffffffffffffff)) > 1e-08)
00001300                    result = 0;
000012d3                else  // gets absolute value
000012d3                {
000012d5                    rax_1 += 1;
000012d5
000012dd                    if (rax_1 == 8)
000012dd                    {
000012ed                        rdx_1 += 1;
000012f0                        cur_s_buffer_start = &cur_s_buffer_start[8];
000012f0
000012f7                        if (rdx_1 != 8)
000012f7                            break;
000012f7
00001324                        result = 1;
000012dd                    }
000012dd                    else
000012dd                        continue;
000012d3                }  // gets absolute value
000012d3
0000130d                *(uint64_t*)((char*)fsbase + 0x28);
0000130d
00001316                if (rax == *(uint64_t*)((char*)fsbase + 0x28))
00001323                    return result;
00001323
0000132b                __stack_chk_fail();
0000132b                /* no return */
000012df            }
```

The second part of the code just iterates over the `s` buffer that is created in the start of the function and checks whether the absolute values are smaller than `1e-08`. Since we seem to be dealing with floats and the value being so small, it is most likely the bounds of a rounding error that can arise with floating point arithmetic, and theoretically all the values that it checks should be zero in order for this loop to return a true value.

Now, lets look at how this array `s` is built in the first part of the function.
The decompilation is a bit messy, but hopefully the parameter names make ti a bit clearer as to what is going on, but what it seems to be doing is taking the user provided buffer that was modified some way in the main function, and then calculates an array of values based on that and the what i have called the flag buffer given inside the binary. This is very messy, so let's rewrite this in python and reverse that instead, since it would be much easier to understand.

The function I finally created after trying to replicate it one to one and then cleaning up a bit of messiness by replacing them with more normal for loops instead of while loops I ended up with this:

```python
def check_validity(input_buf: list[float], flag_buf: list[float]) -> bool:
    s_buf = [0.0 for i in range(64)]
    s_buf_idx = 0

    for input_buf_idx in range(0, 64, 8):
        for flag_buf_start_idx in range(8):
            dbl_val = s_buf[s_buf_idx] # Initially zero
            flag_buf_idx = flag_buf_start_idx
            for cur_inpt_buf_idx in range(input_buf_idx, input_buf_idx + 8):
                dbl_val += input_buf[cur_inpt_buf_idx] * flag_buf[flag_buf_idx]
                flag_buf_idx += 8

            s_buf[s_buf_idx] = dbl_val
            s_buf_idx += 1

    # Here the result should be an array [1.0, 0.0, 0.0, ...]
    s_buf_idx = 0
    iteration_count = 0
    while True:
        cur_idx = 0

        while True:
            subtractor = 0.0
            if (iteration_count == cur_idx):
                subtractor = 1.0

            big_dbl_val = s_buf[s_buf_idx + cur_idx] - subtractor

            if abs(big_dbl_val) > 1e-08:
                return False
            else:
                cur_idx += 1

                if cur_idx == 8:
                    iteration_count += 1
                    s_buf_idx += 8

                    if iteration_count != 8:
                        break

                return True
```

Looking at this, it seems that each value in s is calculated as the sum of 8 values obtained by multiplying values from the flag buffer and the input buffer together.

This array s is then checked whether the values at indexes `0, 9, 19...` are 1 and all other are 0.

> After solving the challenge and discussing it with a fellow ECSC team Estonia team member, they told me that this is actually matrix multiplication. It is checking whether the user provided buffer that has been modified some way is the inverse of the flag buffer matrix. The loop after that checking whether all the values are 0 or 1 is checking whether the resulting matrix is the identity matrix of size 8. This makes the challenge a lot clearer to understand, but at the time of solving I did not realize this. This writeup will follow the premise of me not knowing that it is matrix multiplication, to indicate what I was thinking at the time of solving.

Since the `flag_buf` array is hardcoded into the binary, we want to find a corresponding `input_buf` that will result in the array that passes the check (i.e an array that has the values `[1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, ...]`).
This is actually the same as solving systems of linear equations. Since the value at each index of s is being calculated by adding up the products of 8 values from the `input_buf` and `flag_buf`, we just need to create a system of linear equations that have 8 variables, where the solutions to 7 of them are 0 and the eight solution is 1.
The index of the equation that must equal 1 increases after every equation system (or 8 values of `s` buffer being filled). Let's write that in python

```
from sage.all import *

FLAG_BUFFER = [3.6301095364090035, 2.3589821931566086, 6.1660805621161501,
1.2239986890293528, -6.0618831662560169, -7.7259443234679273,
-0.25581197387195354, 0.27475051194565742, 4.9401832694561625,
3.209195885184394, 7.701062331914434, 1.2786384788737726,
-7.8448270855730646, -9.8551284199142888, -0.34407690477833791,
0.38021100735559177, 5.9034821849783645, 3.4018256643468017,
8.7972188128819919, 1.4738951728897602, -9.0182738214816744,
-11.294265404838294, -0.34693213979523002, 0.44374119792790678,
-2.7200329217602399, -1.5918203680877663, -3.8538449247206183,
-0.67410002581153772, 3.9608078877216619, 5.2125500339486504,
0.16928996217300549, -0.20322901149426972, 0.40361955732664967,
-0.045878317407418649, 0.39391343831104852, -0.035966062645669622,
-0.51150001790590505, -0.51785963721997486, 0.21873984892260298,
0.036916105250672716, -19.629061620759678, -11.474132717808498,
-29.652350941022434, -4.5280861176936336, 29.909158853399131,
38.006855850148128, 1.138880083388917, -1.5659432214429729,
293.10064692934344, 190.87933914328877, 488.40287568233299,
88.963844222897563, -513.56790126665055, -614.85253250585674,
-27.378388570932476, 89.124135619083148, -286.04323446460381,
-191.30915857115647, -493.03131802777858, -95.397700254206327,
524.14305582126246, 615.48466423276602, 27.241994178506811,
-88.72243249412856 9]
```

```
def solve_linear_equations(input_buf_idx_start: int, flag_buf:
list[float], offset_to_be_1: int):
    answers = []

    x1, x2, x3, x4, x5, x6, x7, x8 = var('x1 x2 x3 x4 x5 x6 x7 x8')

    coeffs = []
    rhs = []
    for flag_buf_start_idx in range(8):
        cur_flag_idx = flag_buf_start_idx
        cur_coeffs = []
        for i in range(input_buf_idx_start, input_buf_idx_start + 8):
            cur_coeffs.append(flag_buf[cur_flag_idx])
            cur_flag_idx += 8
        coeffs.append(cur_coeffs)
        if flag_buf_start_idx == offset_to_be_1:
            rhs.append(1.0)
        else:
            rhs.append(0.0)

    equations = []
    for i in range(8):
        eq = sum(coeffs[i][j] * var(f'x{j+1}') for j in range(8)) ==
rhs[i]
        equations.append(eq)

    solutions = solve(equations, x1, x2, x3, x4, x5, x6, x7, x8,
solution_dict=True)
    answers = list([[s[x1].n(), s[x2].n(), s[x3].n(), s[x4].n(),
s[x5].n(), s[x6].n(), s[x7].n(), s[x8].n()] for s in solutions][0])
    return answers


def build_valid_input_buf(flag_buf: list[float]):
    input_buf = []
    for input_buf_idx in range(0, 64, 8):
        offset = input_buf_idx // 8
        answers = solve_linear_equations(input_buf_idx, FLAG_BUFFER,
offset)
        input_buf.extend(answers)

    return input_buf
```

If we run this buffer through the `check_validity` function we see that it passes. Great success!

Now let's return back to the main function.

```
00001330  int32_t main(int32_t argc, char** argv, char** envp)

00001330  {
0000133f      void* fsbase;
0000133f      int64_t rax = *(uint64_t*)((char*)fsbase + 0x28);
00001352      char input_buffer[0x40];
00001352      __builtin_memset(&input_buffer, 0, 0x40);
000013c9      double changeable_buffer[0x40];
000013c9      __printf_chk(1, "Please tell me the code: ", envp, __builtin_memset(&changeable_buffer, 0, 0x200));
000013e2      fgets(&input_buffer, 0x40, stdin);
000013e7      int16_t or_value = -0x100;
000013e7
0000143f      for (int64_t i = 0; i != 64; )
0000143f      {
00001404          double obfuscated_double = obfuscate((((int16_t)input_buffer[i]) | or_value));
00001409          int32_t usually_i = ((int32_t)(i + 7));
00001409
0000140e          if (i >= 0)
0000140e              usually_i = i;
0000140e
0000141b          uint32_t always_zero_value = ((i >> 31) >> 29);
0000142d          // swaps first 3 bits and last 3 bits
0000142d          changeable_buffer[(((int64_t)(usually_i >> 3)) + (((int64_t)(((always_zero_value + i) & 7) - always_zero_value)) << 3))] = obfuscated_double;
00001432          i += 1;
00001436          or_value -= 0x100;
0000143f      }
0000143f
00001452      if (check_validity(&changeable_buffer, &data_4020) == 0)
```

Let's ignore the call to the `obfuscate` function (named by me) for now and look at how the value it outputs is put into the `changeable_buffer` array, which is what is passed to the `check_validity` function (`changeable_buffer`'s end state is what we just reversed previously).

If we look at it closely, we realize that after the bit shifts all it does it is it swaps the first 3 bits and last 3 bits of the `i` value (assuming `i` is 6 bit number for simplicity). The `usually_i` is just set to 7 when it's 0 so that zeroes arent shifted. This is fairly easy to reverse: just start iterating from the back of the array and swap the first 3 and last 3 bits of the index to obtain the original position the value was in.

The value that is in the `changeable_buffer` is calculated based on the `input_buffer` and an or value. Reversing the or value is simple enough - just set it to the maximum and start iterating from the back, this time increasing it by `0x100` instead of decreasing.

Now it's time to take a look at the `obfuscate` function.

```
000011a9  double obfuscate(int16_t arg1) __pure

000011a9  {
000011ad      int32_t i = 0;
000011b2      int32_t rdx = 0;
000011b7      int32_t rsi = 1;
000011bc      int32_t rcx = 1;
000011c1      int32_t r8 = 0;
000011c1
000011d7      do
000011d7      {
000011dc          // checks if bit in position i is 1
000011dc          if ((TEST_BITD(((uint32_t)arg1), i)))
000011dc          {
000011cc              r8 += rsi;
000011cf              rcx += rdx;
000011dc          }  // checks if bit in position i is 1
000011dc          else
000011dc          {
000011de              rsi += r8;
000011e1              rdx += rcx;
000011dc          }  // checks if bit in position i is 1
000011dc
000011d1          i += 1;
000011d7      } while (i != 16);
000011d7
00001202      return (((double)((uint64_t)(r8 + rsi))) / ((double)((uint64_t)(rcx + rdx))));
000011a9  }
```

This looks too much effort to reverse, but why reverse when you can brute force? Since we know the flag consists of only ascii characters, we just have to try all ascii characters that are binary OR'ed with the or_value to see which one matches to obtain the original character. Let's rewrite this obfuscate function in python along with the function to reverse it in order to finally reverse the whole thing and obtain the flag.

The final solve script looks like this:

```
from sage.all import *

FLAG_BUFFER = [3.6301095364090035, 2.3589821931566086, 6.1660805621161501,
1.2239986890293528, -6.0618831662560169, -7.7259443234679273,
-0.25581197387195354, 0.27475051194565742, 4.9401832694561625,
3.209195885184394, 7.701062331914434, 1.2786384788737726,
-7.8448270855730646, -9.8551284199142888, -0.34407690477833791,
0.38021100735559177, 5.9034821849783645, 3.4018256643468017,
8.7972188128819919, 1.4738951728897602, -9.0182738214816744,
-11.294265404838294, -0.34693213979523002, 0.44374119792790678,
-2.7200329217602399, -1.5918203680877663, -3.8538449247206183,
-0.67410002581153772, 3.9608078877216619, 5.2125500339486504,
0.16928996217300549, -0.20322901149426972, 0.40361955732664967,
-0.045878317407418649, 0.39391343831104852, -0.035966062645669622,
-0.51150001790590505, -0.51785963721997486, 0.21873984892260298,
0.036916105250672716, -19.629061620759678, -11.474132717808498,
-29.652350941022434, -4.5280861176936336, 29.909158853399131,
38.006855850148128, 1.138880083388917, -1.5659432214429729,
293.10064692934344, 190.87933914328877, 488.40287568233299,
88.963844222897563, -513.56790126665055, -614.85253250585674,
-27.378388570932476, 89.124135619083148, -286.04323446460381,
-191.30915857115647, -493.03131802777858, -95.397700254206327,
524.14305582126246, 615.48466423276602, 27.241994178506811,
-88.722432494128569]

def test_bitd(val, i) -> bool:
    return (1 & (val >> i)) != 0


def obfuscate(val):
    rdx = 0
    rsi = 1
    rcx = 1
    r8 = 0

    for i in range(16):
        if test_bitd(val, i):
            r8 += rsi
            rcx += rdx
        else:
            rsi += r8
```

```
            rdx += rcx

    return (r8 + rsi) / (rcx + rdx)


def solve_linear_equations(input_buf_idx_start: int, flag_buf:
list[float], offset_to_be_1: int):
    answers = []

    x1, x2, x3, x4, x5, x6, x7, x8 = var('x1 x2 x3 x4 x5 x6 x7 x8')

    coeffs = []
    rhs = []
    for flag_buf_start_idx in range(8):
        cur_flag_idx = flag_buf_start_idx
        cur_coeffs = []
        for i in range(input_buf_idx_start, input_buf_idx_start + 8):
            cur_coeffs.append(flag_buf[cur_flag_idx])
            cur_flag_idx += 8
        coeffs.append(cur_coeffs)
        if flag_buf_start_idx == offset_to_be_1:
            rhs.append(1.0)
        else:
            rhs.append(0.0)

    equations = []
    for i in range(8):
        eq = sum(coeffs[i][j] * var(f'x{j+1}') for j in range(8)) ==
rhs[i]
        equations.append(eq)

    solutions = solve(equations, x1, x2, x3, x4, x5, x6, x7, x8,
solution_dict=True)
    answers = list([[s[x1].n(), s[x2].n(), s[x3].n(), s[x4].n(),
s[x5].n(), s[x6].n(), s[x7].n(), s[x8].n()] for s in solutions][0])
    return answers


def build_valid_input_buf(flag_buf: list[float]):
    input_buf = []
    for input_buf_idx in range(0, 64, 8):
        offset = input_buf_idx // 8
        answers = solve_linear_equations(input_buf_idx, FLAG_BUFFER,
offset)
        input_buf.extend(answers)

    return input_buf


def check_validity(input_buf: list[float], flag_buf: list[float]) -> bool:
```

```
    s_buf = [0.0 for i in range(64)]
    s_buf_idx = 0

    for input_buf_idx in range(0, 64, 8):
        for flag_buf_start_idx in range(8):
            dbl_val = s_buf[s_buf_idx] # Initially zero
            flag_buf_idx = flag_buf_start_idx
            # Get large linear equation system here that needs to be solved to obtain the values
            for cur_inpt_buf_idx in range(input_buf_idx, input_buf_idx + 8):
                dbl_val += input_buf[cur_inpt_buf_idx] * flag_buf[flag_buf_idx] # this needs to be minimized for cases where s_buf_idx != 0, 9, 19, 30, 42 ..., else it needs to be as close to 1 as possible
                flag_buf_idx += 8

            s_buf[s_buf_idx] = dbl_val
            s_buf_idx += 1

    # Here the result should be an array [1.0, 0.0, 0.0, ...]
    s_buf_idx = 0
    iteration_count = 0
    while True:
        cur_idx = 0

        while True:
            subtractor = 0.0
            if (iteration_count == cur_idx):
                subtractor = 1.0

            big_dbl_val = s_buf[s_buf_idx + cur_idx] - subtractor

            if abs(big_dbl_val) > 1e-08:
                return False
            else:
                cur_idx += 1

                if cur_idx == 8:
                    iteration_count += 1
                    s_buf_idx += 8

                    if iteration_count != 8:
                        break

                return True


def main_func(inpt: str):
    or_val = -0x100
```

```python
    inpt_buf = [ord(i) for i in inpt]
    changeable_buffer = [0.0 for i in range(64)]
    for i in range(64):
        dbl = obfuscate(inpt_buf[i] | or_val)

        usually_i = i + 7 # Guarantees that all bits aren't zero
        if i >= 0:
            usually_i = i

        # Swap first and last 3 bits
        changeable_buffer[(usually_i >> 3) + ((i & 7) << 3)] = dbl

        or_val -= 0x100

    if check_validity(changeable_buffer, FLAG_BUFFER):
        print("found flag!")
        print(changeable_buffer)
    else:
        print("Wrong")


def reverse_obfuscation(changeable_buffer: list[float]):
    or_val = -0x100
    for i in range(63):
        or_val -= 0x100

    original_inpt_buf = [-1 for i in range(64)]
    for i in range(63, -1, -1):

        idx = i
        if i == 0:
            idx = 7

        idx = (idx >> 3) + ((i & 7) << 3)
        obfuscated_val = changeable_buffer[idx]
        for j in range(128):
            if abs(obfuscate(j | or_val) - obfuscated_val) > 1e-08:
                continue
            original_inpt_buf[i] = j
            break
        or_val += 0x100

    return ''.join(chr(i) for i in original_inpt_buf)

valid_inpt_buf = build_valid_input_buf(FLAG_BUFFER)
flag = reverse_obfuscation(valid_inpt_buf)

print(flag)
```