EsoVM

Abstract

This is a very interesting challenge: an interpreter that execute `Esolang` is implemented, and within that `Esolang` another VM is also implemented, so this is a double VM challenge. Within that VM, some finite field arithmetic operations are performed on the input flag, whose result is compared with some constants, and the `correct` message will be printed if they are equal.

Esolang Interpreter

The `Esolang` in this challenge is a little bit similar to `BrainFuck`, but some extended operations are supported. It is clear if you look at the code of interpreter, which is easy to understand because the symbol is even not stripped out. We can slightly change the code of interpreter, and make it a C code generator that transform the `Esolang` back to the C code.

Esolang Code

The Esolang code being generated is huge. However, many previous lines are simply the lines that are used to generate constants. These are not important and we don't need to understand them, but we do need the result from these operations. Therefore, we can set a breakpoint at the beginning of first while loop, and dump the memory data, `reg` and `p` (memory data is at `0x204040`). In this way we can remove the previous lines used to generate constants, and substitute them with a file read operation. The result source code is here.

Reverse Engineering the Esolang C Codes

However, such codes are not readable and hard to understand, which got me stuck for many hours. However, we can see several patterns.

```
mem[p] = (mem[p] + 0x10000) % 65537; //-1
while ( mem[p] )
{
    mem[p] = (mem[p] + 1) % 65537;
    ++p;
    ++p;
    ++p;
    ++p;
    ++p;
    ++p;
    ++p;
    mem[p] = (mem[p] + 0x10000) % 65537;
}//find first mem[p] == 1 for field0
```

```
mem[p] = (mem[p] + 0x10000) % 65537;
while ( mem[p] )
{
    mem[p] = (mem[p] + 1) % 65537;
    --p;
    --p;
    --p;
    --p;
    --p;
    --p;
    --p;
    mem[p] = (mem[p] + 0x10000) % 65537;
}//switch to field1, find first mem[p] == 1
```

These are the codes that try to find the first memory block with value `1`, and we can also see that the structure size is `7*8==0x38`.

```
mem[p] = reg; //field6, 2field1
while ( mem[p] )
{// reg == 4 => 3 2 1 0 0
    mem[p] = (mem[p] + 0x10000) % 65537;
    reg = mem[p];
    ++p;
    ++p;
    ++p;
    ++p;
    mem[p] = reg;
}
```

These are the codes that is used to achieve the effect of index accessing, something like `p[reg]`. Here the structure size is `4*8 == 0x20`.

```
mem[p] = mem[p] == mem[p + 1]; //field1 == field2 0x03
while ( mem[p] )
{
    mem[p] = (mem[p] + 0x10000) % 65537;
    //...
}
```

Also sometimes the while loop is used as a `if` as shown above. This actually implements a `switch case` statement that run different handler for different `opcode`.

Understanding what these loops actually are doing and inspecting the memory being dumped, we can find that a virtual machine is implemented using Esolang. The memory is separated into 2 segments: first segment is codes, and second segment is data. The size of structure in code segment is `0x38`, and the size of structure in data segment is `0x20`. The structure is shown below.

```
//size of all fields is 8
//data
2field0=1field5 mark, always 1 for first element
```

```
2field1 4 => 3 2 1 0 0
2field2 increment for given field3 as idx
2field3 [field3].2field3 = [field4].2field3

//instruction
field0 last(152) mark
field1 first(i) mark
field2 field3 0?
field4 opcode
field5 field6 arg?

//at index 152
last.field2 = this.field4 //opcode
last.field3 = this.field5 //arg1
last.field4 = this.field6 //arg2
```

The structures are shown above.

I think I need to talk about `field0` and `field1` of the instruction a bit. `field0` is always 0 for all opcodes, except for the data structure at index `152` it is `1`, which is like a "bridge" between codes and data. When the Esolang program need to let `p` point to data, it will use the loop I mentioned above to locate that bridge which is just before the data segment. `field1` is `1` only when Esolang VM is executing that instruction; in other word it is another way to implement the program counter. This is needed to let `p` to point back to the current instruction being executed when `p` is at the data segment.

At index 152, its `field0` is 1 and `field1` is 0, which are used to navigate to the "bridge" and back to the current instruction being executed, as I mentioned above. `field2/3/4` are used to retrieve the argument of current instruction, they can be regarded as temporary register maybe? And starting from `field5`, it becomes the array of `data`. To be specific `last.field5` is same as `data[0].field0`.

For `data`, `2field0` is the mark of the "bridge" (`2` is only used to distinguish `data` and `instruction`), just like `field0` of the "bridge" that I mentioned above. `2field1` is used to implement the index accessing, which was also mentioned above. `2field2` will increment at `data[reg]` when `reg` is used as index to access the data segment, not sure how is this useful. `2field3` is the most important field, which holds the actual data at this memory block. `[arg1].2field3 = [arg2].2field3` is the operation that will be done for `0x03` opcode.

Then after some reversing, we can write the disassembler. (This also took long time :D)

```python
from struct import unpack

u64 = lambda x : unpack('<Q', x)[0x0]
f = open("dump.bin", "rb")
data = f.read()
f.close()

asm = []
for x in xrange(0, 152*7*8, 7*8):
    tmp = []
    for i in xrange(x, x+7*8, 8):
        tmp.append(u64(data[i:i+8]))
    asm.append(tmp)
```

```
ram = []
for i in xrange(152*7*8+8*5, len(data) / 8 * 8, 8):
    ram.append(u64(data[i:i+8]))

def disasm(asms):
    ret = []
    for asm in asms:
        opcode = asm[4]
        if opcode == 0x00:
            if asm[5] == 0x00:
                ret.append("[0x0] += [0x1]")
            elif asm[5] == 0x01:
                ret.append("[0x0] *= [0x1]")
            elif asm[5] == 0x02:
                ret.append("[0x0] = [0x0]==[0x1]")
            elif asm[5] == 0x03:
                ret.append("[0x0] = getchar()")
            elif asm[5] == 0x04:
                ret.append("putchar([0x0])")
            else:
                assert(False)
        elif opcode == 0x01:
            ret.append(str(asm[6]) + ' ' + str(asm[5]))
        elif opcode == 0x02:
            ret.append("[[%s]] = [[%s]]" % (hex(asm[5]), hex(asm[6]))) #todo
        elif opcode == 0x03:
            ret.append("[%s] = [%s]" % (hex(asm[5]), hex(asm[6])))
        else:
            assert(False)
    return ret
```

One thing to note is `opcode 0x02`, in which the `last.field2` is incremented finally, and handler for `0x03` will thus also be executed.

```
mem[p] = (mem[p] + 1) % 65537; //lfield2++, execute 0x03
```

But now `last.field3` and `last.field4` are already the data fetched from the `data`, which are `data[arg1]` and `data[arg2]`, so this is a double memory access instruction.

The output is here.

Reverse Engineering the VM Code

Finally we have the VM code. This is not hard to understand, but you may always need to refer to the dumped memory for constants. There are basically 3 parts in this code: encoding the input flag, comparing the result of encoding with a constant array, and printing the message according to the result of comparison.

Translating the encoding to Python, we have this.

```
arr = lambda i : ram[4*i+3]
def encode1(flag):
    r = [0] * 0x22
    flag = map(ord, flag)
    for y in xrange(0x22):
        c = flag[y]
        last = 0
        for x in xrange(0x22):
            c = (((c * arr(y + 8)) % 65537) + arr(y + 0x2c)) % 65537
            r[x] = (r[x] + c) % 65537
    return r
```

To write it mathematically, we have $r_i = \sum_{j=0}^{33} c_{(i+1)j}$ (note `0x22-1 == 33`), where c_{0j} is `flag[j]` and $c_{(i+1)j} = c_{ij} \times a_j + b_j$, a_j is `arr(j + 8)`, b_j is `arr(j + 0x2c)`, and r_i is the result of encoding, `r[i]`. Note that the add and multiplication here are all finite field over 65537.

Solving the Equation

Well, we have already got the way to encode the flag, and we know the result of encoding (starting from data index `0x76`), but how to solve this to get original flag? Since one character of flag will contribute to all elements of the result, we cannot brute force byte by byte, and brute force 34 bytes does not seem to be possible. My approach is to convert it into a finite field matrix linear equation.

Note that:

$c_{1j} = c_{0j} \times a_j + b_j$

$c_{2j} = c_{1j} \times a_j + b_j = (c_{0j} \times a_j + b_j) \times a_j + b_j = c_{0j} \times a_j^2 + b_j \times a_j + b_j$

$c_{3j} = c_{0j} \times a_j^3 + b_j \times a_j^2 + b_j \times a_j + b_j$

To generalize it:

$c_{ij} = c_{0j} \times a_j^i + b_j \times \sum_{k=0}^{i-1} a_j^k$

Putting it into the equation that calculates r:

$r_i = \sum_{j=0}^{33}(c_{0j} \times a_j^{i+1} + b_j \times \sum_{k=0}^{i} a_j^k) = \sum_{j=0}^{33}(c_{0j} \times a_j^{i+1}) + \sum_{j=0}^{33}(b_j \times \sum_{k=0}^{i} a_j^k)$

Now, we can subtract every r_i by $\sum_{j=0}^{33}(b_j \times \sum_{k=0}^{i} a_j^k)$, which is known, and we get $\sum_{j=0}^{33}(c_{0j} \times a_j^{i+1})$, denoting as e_i.

This seems to be familiar, which is the formula of matrix multiplication by a vector :D

Writing it in matrix form, we have $E = Ac_0$, where $A = \begin{bmatrix} a_0^1 & a_1^1 & a_2^1 & \dots \\ a_0^2 & a_1^2 & \dots & \\ a_0^3 & a_1^3 & \dots & \\ a_0^4 & a_1^4 & \dots & \\ \dots & & & \end{bmatrix}$, which is 34 by 34 square matrix.

Note that all operation, including subtraction and power, are in finite field 65537.

Then we need to find a tool to solve this finite field linear matrix equation, but this is where I gave it up in contest because it is almost 4 a.m, and I could not find proper tool to solve it. Later on, I found this, which is a tool that support finite field matrix equation solving. Finally, this is the script that solves everything.

`TSGCTF{vm_On_vm_c4n_be_r3ver5able}`