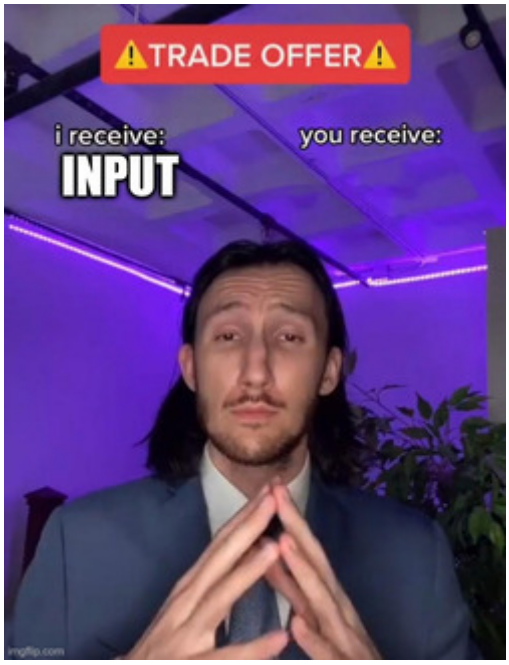
VMV

Intro

VMV was a 500pts (499pts, 14th solves at the moment where I'm writing this WU) reverse-engineering challenge at the FCSC 2021.
The executable file is an ELF x64.

First thing first, we try to run the elf, which is asking for a 16-bytes string as an argument. If we give it a valid string, it enters a loop telling us "Do not leave, a correspondent will take your call ... ".

With my huge experience of French SAV services I already knew it was going to take forever and that it was probably not worth it to try anything dynamically.



First look

I then loaded the binary in IDA, and saw an hideous monster:

```
qword_A310 = (__int64)calloc(8uLL, 0x20000uLL);
sub_1238(11388205LL, sub_14CD);
sub_1238(90206877LL, sub_1652);
sub_1238(1220922566LL, sub_1438);
sub_1238(1411387552LL, sub_1590);
sub_1238(3186425429LL, sub_17BB);
sub_1238(1106852683LL, sub_17D4);
sub_1238(1583659884LL, sub_17EA);
sub_1238(3981323515LL, sub_1835);
sub_1238(403423533LL, sub_1866);
sub_1238(1510947068LL, sub_18F9);
sub_1238(659126534LL, sub_198C);
sub_1238(3121682089LL, sub_19A2);
sub_1238(4202953310LL, sub_19DE);
sub_1238(2173490869LL, sub_1A1D);
sub_1238(2372385505LL, sub_1A64);
sub_1238(3510963559LL, sub_1ABF);
sub_1238(2393135928LL, sub_1B46);
sub_1238(4027299521LL, sub_1B03);
sub_1238(1502722594LL, sub_1B89);
sub_1238(1135484755LL, sub_1724);
sub_1238(2304805002LL, sub_1BE5);
sub_1238(520785519LL, sub_1C41);
sub_1238(1181373657LL, sub_1C8E);
sub_1238(4216461980LL, sub_1CA4);
sub_1238(3327193437LL, sub_1CF7);
src = (void *)sub_20C6(aLwmaaai7q4caba, 16144LL);
v4 = strlen(a2[1]);
dest = (char *)calloc(1uLL, v4 + 12108);
v6 = sub_20C6(aQryrugaeaaaiup, 4904LL);
memcpy(dest, src, 0x2F4CuLL);
v7 = strlen(a2[1]);
memcpy(dest + 12108, a2[1], v7);
v11 = sub_12BC(v6, dest);
puts("[fr] Ne quittez pas, un correspondant va prendre votre appel... [\\fr]");
v12 = 0LL;
while ( *(_BYTE *)(v11 + 128) )
{
  v8 = sub_137F(v11);
  v9 = (void (__fastcall *)(__int64))sub_11D5(v8);
  v9(v11);
  if ( !(++v12 & 0xFFFFFFF) && *(_BYTE *)(v11 + 136) != 1 )
    puts("[fr] Ne quittez pas, un correspondant va prendre votre appel... [\\fr]");
}
return 0LL;
```

Function table

Luckily for us, the function `sub_1238` is pretty simple:

```
_DWORD *__fastcall sub_1238(int a1, __int64 a2)
{
  int v2; // STOC_4
  _DWORD *result; // rax

  v2 = a1;
  result = calloc(0x18uLL, 1uLL);
  result[2] = a1;
  *((_QWORD *)result + 2) = a2;
  *(_QWORD *)result = *(_QWORD *)(qword_A310 + 8LL * (v2 * v2 & 0x1FFFF));
  *(_QWORD *)(8LL * (v2 * v2 & 0x1FFFF) + qword_A310) = result;
  return result;
}
```

And luckily for me, I recognized a structure that I saw two weeks ago in algorithmic classes: it's an hashtable.

`a1` and `a2` are the parameters given to the function, respectively: the function key, and the function address.

We can see that `sub_1238` put the function address and the function key in a struct:

```
result[0] = qword_A310[key * key & 0x1ffff]
result[1] = a2
result[2] = a1
```

Then, the new struct is put into `qword_A310`, which is our hashtable:

```
qword_A310[a1 * a1 & 0x1ffff] = result
```

We can clearly see that the hash function here is: `hash(key) = key * key & 0x1ffff`

The fact that `result[0] = qword_A310[hash(key)]` indicate us that there might be hash collisions here, meaning that if two keys have the same hashes, the new structure will have a pointer to old one and then take it's place into the hashtable. In the end, we'll have a linked-list of functions structures.

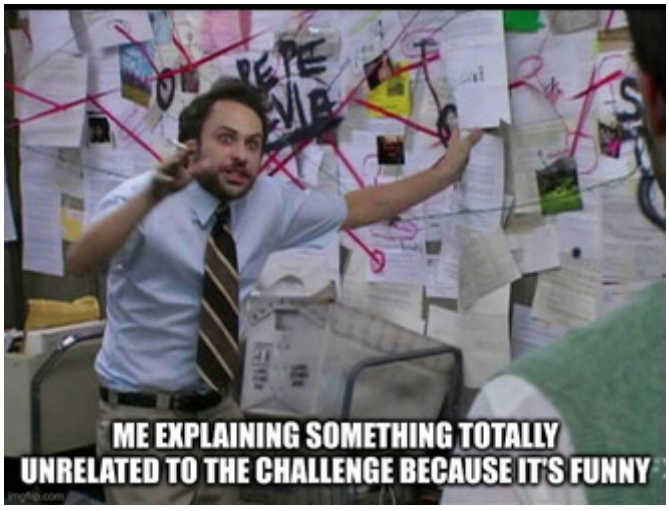
To illustrate this, we can study another function related to this hashtable: `sub_11D5`

```
__int64 __fastcall sub_11D5(int a1)
{
  __int64 *i; // [rsp+Ch] [rbp-8h]

  for ( i = *(qword_A310 + 8LL * (a1 * a1 & 0x1FFFF)); i; i = *i )
  {
    if ( a1 == *(i + 2) )
      return i[2];
  }
  return 0LL;
}
```

This function take a key as an argument, and return the appropriate function. We can clearly see how an hashtable works here: we compute the hash of the key `a1` and we get the structure linked to that hash, if the key put as an argument and the key stored in the structure doesn't match, we then get the structure linked by the first one, and compare the key again, we do that over and over, until we find the appropriate structure which hold the function pointer we're looking for.

Note that this isn't really important for the challenge, I just found it funny to see an application of hashtables just after having this class at college, and I thought this would be a nice exercise to try to explain how this works.



Data extraction

We then fall on `sub_20C6`, the main function give two parameters to it: a base64-looking string and it's size. I gave a quick look to this function and assumed that this was a base64 decoding routine and skipped it.

I then extracted the two base64-encoded buffers with IDA:

```
import base64
open("part1.bin").write(base64.b64decode(idaapi.get_bytes(0x50a0, 16144)))
open("part2.bin").write(base64.b64decode(idaapi.get_bytes(0x8fc0, 4904)))
```

Running `file` on them doesn't seem to give us anything, which isn't really surprising. These two buffers are really important, but I'll come back to them later.

Smol things

We have 2 interesting functions left: `sub_12BC` and `sub_137F`.

- `sub_12BC` is a function initializing an array with 64-bits & 32-bits values so it's pretty likely to be another struct. Note that it takes our two base64-decoded buffers in argument and stores pointers to it in two structs members.

```
_QWORD *__fastcall sub_12BC(__int64 a1, __int64 a2)
{
  _QWORD *v2; // ST18_8

  v2 = calloc(0x90uLL, 1uLL);
  *v2 = calloc(4uLL, 0x400uLL);
  v2[12] = calloc(4uLL, 0x2000000uLL);
  v2[11] = a1;
  v2[14] = a2;
  v2[15] = 0LL;
  *(v2 + 128) = 1;
  *(v2 + 136) = 0;
  return v2;
}
```

- `sub_137F` is a function taking the result of `sub_12BC` returning a 32-bit value:

```
__int64 __fastcall sub_137F(__int64 a1)
{
  return *(*(a1 + 88) + 4LL * (*(a1 + 36))++);
}
```

  We recognize that `*(a1+88)` is actually the `v2[11]` of `sub_12BC`'s result, so this is our second extracted buffer. It's being indexed by `*(a1+36)` which is initialized to zero and being incremented after usage. The function might look obscure in the first place, but it becoming really clear once we put everything together.

After all these, the function we got through `sub_11D5` is executed with the result of `sub_12BC` as a parameter. Then we have an index getting incremented and the `Do not leave, a correspondent will take your call ...` being printed if the index is a multiple of `0x100000000` (and if another variable is different from -1, but it's not really interesting, it's just a check to be sure the virtual machine code printed nothing on screen before), so we can fear that this loop is getting executed a lot of times.

Everything together !

Putting all together we reach the following conclusions:

- The hashtable is actually the instruction set of the virtual machine
- `sub_12BC` is generating the virtual machine's state (VMS)
- `sub_137F` is incrementing the instruction pointer and giving us the instruction to execute
- the struct member being used as instruction pointer is `*(a1+36)`
- the code running in the VM is the file `part2.bin` we extracted

All we have to do now is reversing every function in the hashtable to understand which instruction code correspond to which action.

VM instructions

I'm not gonna details every instructions, just explains a few of them which illustrates pretty well of the VM's internals works. I think it's important to take time to understand how this VM works because it's very important to have a great understanding of it for the rest of the challenge.

First thing to know, is that we have to important functions that are being called in almost every other one:

```
__int64 __fastcall sub_13FD(__int64 a1)
{
  return *(*(a1 + 88) + 4LL * (*(a1 + 36))++);
}
```

```
__int64 __fastcall sub_13BA(__int64 a1)
{
  int v1; // ST14_4

  v1 = *(*(a1 + 88) + 4LL * (*(a1 + 36))++);
  return (v1 ^ 0x7B) & 0x3F;
}
```

Those two functions are really similar to the function `sub_137F`, and in fact they're doing almost the same thing. The first one, `sub_13FD` is used to get a word from program memory which will be used in every case as an immediate value. The second one, `sub_13BA` get a word from the very same memory but then xor it, it will be used by instructions to get the register to use.

In the rest of the article, i'll write `imm` for raw 4-bytes word and `reg` for an unknown register.

Push and stack instructions

Understanding how the stack works

I think the best way to explain how the VM's stacks works is to show the `push reg` instruction.

```
__int64 __fastcall sub_1B89(__int64 a1)
{
  __int64 v1; // ST00_8
  unsigned int v2; // ST10_4
  __int64 result; // rax

  v1 = a1;
  v2 = *(a1 + 4LL * sub_13BA(a1) + 8);
  ++*(v1 + 56);
  result = v2;
  *(4LL * *(v1 + 56) + *v1) = v2;
  return result;
}
```

(result is a variable generated by IDA, here we can just ignore it as the function isn't supposed to return anything, if you wanted to get a cleaner code in IDA, you can just retype the function and cast it to void)

As we can see, the first thing the instruction does is getting the next register with

```
v2 = *(a1 + 4 * sub_13BA(a1) + 8)
```

I explained it before, `sub_13BA` is getting the next word and xoring it, and this line get the corresponding register from the the VMS.

We can see here that we're accessing VMS base address + 8 bytes, which is normal since the 8 first bytes of the VMS are the address pointing to `calloc(4uLL, 0x400uLL)`. We can also make another observation: the register offset we get through `sub_13BA` is multiplied by 4, meaning each register is 32-bits.

If we go back to `sub_12BC` - the function generating the VMS - we can see that the next index being occupied by an 8-bytes address is `v2[11]`, se we can guess that every registers are between `v2[1]` and `v2[11]`, which makes 11 - 1 = 10 64-bits registers, but we already know registers are 32-bits so that gives us 20 32-bits registers.

Knowing all that, we can look at the next line: `++*(v1 + 56)` . `v1` here is a pointer to the VMS, so with our precedent remarks we can easily deduce that is register `(56 - 8) / 4 = 12`. So here, we are incrementing reg12.

Next line is: `*(4LL * *(v1 + 56) + *v1)    v2`, we are accessing a 4-byte word from `*v1`, and we know that `*v1 = calloc(4uLL, 0x400uLL)`. We can translate this as: `v1[0][4 * reg12] = reg`

Let's recap:

- we increase reg12 by one
- we store a register value at `v1[0][4* reg12]`

It's pretty clear: `v1[0]` is the stack, and reg12 plays the role of a stack pointer. We can confirm this by looking at another function which clearly plays the role of a pop instruction.

Other stack instructions

If we continue to reverse-engineer the virtual machine, we'll see something: all the operations are made on the stack. Let's take an example:

```
void __fastcall xor_stack(vms *vms)
{
  vms *vms_; // ST00_8
  int p; // ST14_4
  int q; // ST10_4

  vms_ = vms;
  p = *&vms->stack[4 * vms->reg12--];
  q = *&vms_->stack[4 * vms_->reg12--];
  *&vms_->stack[4 * ++vms_->reg12] = q ^ p;
}
```

(this code is from my original IDA session, where I took time to cleanup everything)

Here this function is playing the role of a xor, we can see that the value `p` and `q` are getting popped (we read value from the stack at reg12 before decreasing it). And then, the result `p ^ q` is getting pushed.

And it's the same for every other arithmetic/bitwise operations: `add`, `and`, etc... It's also the case for conditional branching operations: by example, the instruction `jz` pop two values from the stack and jump if they are equals.

Accumulator instructions

Let's take a look at this function:

```
__int64 __fastcall sub_1352(__int64 a1, int a2)
{
  unsigned int v2; // ST1C_4

  v2 = *(a1 + 104);
  *(a1 + 104) += a2;
  return v2;
}
```

We already knew that we have 20 32-bits registers starting at offset 8, so `*(a1+104)` isn't a regular register as 104 > 8 + 20 * 4 = 88. In fact, this structure come after two others struct members: the pointer to our code, and the pointer to the memory allocated by `calloc(4uLL, 0x2000000uLL)`. Seeing this usage, we can deduce it is some sort of accumulator (according to wikipedia: "In a computer's central processing unit (CPU), the accumulator is a register results are stored.").

Here we have only two instructions using this accumulator:

- `add acc, reg`
- `add acc, imm`

By analyzing these instructions code, we can see that reg8 is used to store the old value of the accumulator before it gets modified. More simply:

```
reg8 = acc
acc += regX
```

Branching instructions

We already saw in the stack's part that the VM features two conditional jump instructions: `jz` and `jnz`. It also has a `jmp imm` and more importantly a `call` and `ret` instructions.

```
void __fastcall call(vms *a1)
{
  __int32 v1; // eax

  v1 = next_word(a1);
  a1->reg9 = a1->reg7;
  a1->reg7 += v1;
}
```

(Again, this is from my original IDA session)

This is the decompiled code of the `call` instruction. We can clearly see that `reg7`, which is the instruction pointer, is getting incremented by an immediate value, and that the return value is stored in `reg9`. Obviously, the `ret` instruction restore `reg7` with the `reg9` instruction.

Note that this is `reg7` being put into `reg9` and not `reg7 + 4`, this is because once the register is restored, at the beginning of the second loop `reg7` will be increased anyway and we'll have access to the next instruction.

Memory instructions

I think this isn't necessary to explain in details how the others functions works, it's the same methodology over and over: you look at the registers involved and deduce how the VM is behaving.

The VM has 3 different memory-related instructions:

- `ldr reg`: this instruction load a word from memory (the `calloc(4uLL, 0x2000000uLL)` one) at the offset `reg8` and store it into `reg`
- `str reg`: this is the opposite operation, we store `reg`'s value into the memory at the offset `reg8`
- `li reg`: this one is slightly different, it does not use the memory but the `data` section (which is our file `part1.bin`) it uses a specific pointer which isn't a register and which is getting incremented after each reading operation.

Others

We have two others functions:

- `hlt`: which stops program execution
- `putchar reg`: which print the value in `reg` and set the VMS's structure member that disable the printing of the `Do not leave, a correspondent will take your call ...` string.

Knowing all that, we just have to implement a python disassembler for this virtual-machine.

The hard truth

We then runs our disassembler and get the code:

```
--- snip snip ---
00000046: push 0x2f206b0b
00000048: jz 0x32e
0000004a: push r11
0000004c: push 0xb8ea3ec1
0000004e: jz 0x2f3
00000050: push r11
00000052: push 0x371ee5a6
00000054: jz 0x35c
00000056: push r11
00000058: push 0x5277de4d
0000005a: jz 0x1ba
0000005c: push r11
0000005e: push 0x1a08a46e
00000060: jz 0x336
00000062: push r11
00000064: push 0xf82ba570
00000066: jz 0xd6
00000068: push r11
0000006a: push 0xa7f8d895
0000006c: jz 0x111
0000006e: push r11
00000070: push 0xca7dcd4e
00000072: jz 0x260
00000074: push r11
00000076: push 0x6de1c5fd
00000078: jz 0x23f
0000007a: push r11
0000007c: push 0x16be1ae2
0000007e: jz 0x1dc
--- snip snip ---
```

And we get horrified by the hard truth: there is a VM inside the VM !

VM-ception

New VM

If I spent that much time explaining how to reverse the first VM it's mostly because the second one behave the same. However, we can note some differences:

- instructions are no longer stored in an hashtable (rip algorithmic classes) but just compared one by one in a jump table as shown in the last code picture.
- the next code is read from `part1.bin`
- it no longer print chars from register, it loads an immediate value and xor it with some key

I reversed this second VM and built another disassembler to see this:

```
--- snip snip ---
00000038: jmp 0x25
0000003a: push 0x3d039463
0000003c: pop r13
0000003e: push 0x4a0cbe95
00000040: pop r9
00000042: call 0x32c
00000044: push r3
00000046: push 0xab243ec5
00000048: jz 0x32e
0000004a: push r3
0000004c: push 0x437919f9
0000004e: jz 0x2f3
00000050: push r3
00000052: push 0xd077bb5e
00000054: jz 0x35c
00000056: push r3
00000058: push 0x87d05da9
0000005a: jz 0x1ba
0000005c: push r3
0000005e: push 0x8bf4943f
```

```
00000060: jz 0x336
00000062: push r3
00000064: push 0xd365d099
00000066: jz 0xd6
--- snip snip ---
```

We have another VM inside ! Great !

part1.bin file format

We know that this knew VM was read from `part1.bin`, from 0 to 0x397. We see that there's still a lot of unused data after the offset 0x397 into `part1.bin` so we can be pretty sure we have another VM inside this one. I like to reverse things but there's a limit: there's no way I reverse all these VMs one by one by hand. So I saw something interesting: the first and second code we disassembled have different instructions, use different registers but the function have exactly the same offsets. Which means, if the offset `0x27b` (random) simulate the `push` instruction for the next VM, the disassembled code of the next VM will simulate `push` instruction at this same offset.

Moreover, the file `part1.bin` we extracted have a specific format: it stores the size of the code before the code. By example it begins like this:

```
0000:0000 97 03 00 00  02 3B AB 87  00 04 00 00  9B 54 D7 3B  .....;«.....Tx;
```

Here we can see that before the first instruction we have 0x0397, which is the size of the code.

I just had to wrote an automated disassembler that will loop through the file and generate the disassembly of each VM. In the end, my script disassembled 4 codes: three of size 0x397, and one of size 0x10a. As the one with the size 0x10a come at the end of the file, we can safely assume it isn't a VM (and the code extracted will prove us right). So that make: 3 VMs + the one the file `part1.bin` + the original: 5 VMs !

this meme was originally made in French and I'm totally too lazy to make it again in English

Last man standing

Now that we have access to the last disassembled program the reverse is over, in the great tradition of the FCSC it had to end with some cryptography.

I'm kind of exaggerating here, as the crypto part wasn't that hard (it's not even cryptography at this point).

The key we enter is splitted in 4 4-bytes part:

- the second part must respect:
  - `k2 % 0x77f3 = 0x4926`
  - `k2 % 0x7c49 = 0x3159`
- the fourth part must respect:
  - `k4 % 0x77f3 = 0x28b2`
  - `k4 % 0x7c49 = 0x44a9`

Knowing that `k2` and `k4` must contains printable characters we can easily brute force them we a C program, which gives us:

- `k2 = 0x30546445 '0TdE'`
- `k4 = 0x72337033 'r3p3'`

Finding `k1` and `k3` took me a little bit longer. To check those two the program uses a function of the very first VM:

```
void __fastcall crypt_stack(regs *regs)
{
  regs *v1; // ST00_8
  int p; // ST14_4
  int q; // ST10_4

  v1 = regs;
  p = *&regs->stack[4 * regs->reg12--];
  q = *&v1->stack[4 * v1->reg12--];
  ++v1->reg12;
  *&regs->stack[4 * regs->reg12] = q * p
                                 - 0x7FFFFFFF
                                 * (((0x200000005LL * (q * p) >> 64) + ((q * p - (0x200000005LL * (q * p) >> 64)) >> 1)) >> 30);
}
```

I didn't understood anything of what this do so I googled the constant `0x200000005LL`, I got really lucky because the search returned only 1 result: HITCON CTF 2017 Quals.

Reading this article we learn that this ugly bit manipulation is just a multiplication mod `0x7FFFFFFF`. In the disassembly we see that the result of this multiplication is compared to 1. So we just have to compute the modular inverse of `0x117052c0` mod `0x7FFFFFFF` for `k1`, and the modular inverse of `0x278bce9d` for `k2`.

We get:

- `k1 = 'eN3w'`
- `k2 = 'r3p3'`

Putting everything together we get:

`FCSC{W3NeeDT0gODe3p3r}`