BattleChip (Misc - 500 points) - bluesheet

Challenge description (translated with the help of DeepL for precision purposes)

We have set up an online CHIP-8 emulator. It takes as input a hexadecimal string representing the ROM to execute. We give more details about the architecture here: https://www.france-cybersecurity-challenge.fr/chip8s

We advise you to start by solving `Chip & Fish`. For this second challenge, your goal is to find and exploit the vulnerability of this architecture allowing you to discover the generated secret.

`nc challenges1.france-cybersecurity-challenge.fr 7004`

The flag is in the form `FCSC{secret.hex()}`.

Chip-8 overview (adapted from Wikipedia)

`CHIP-8` is an **interpreted programming language**, developed by Joseph Weisbecker.

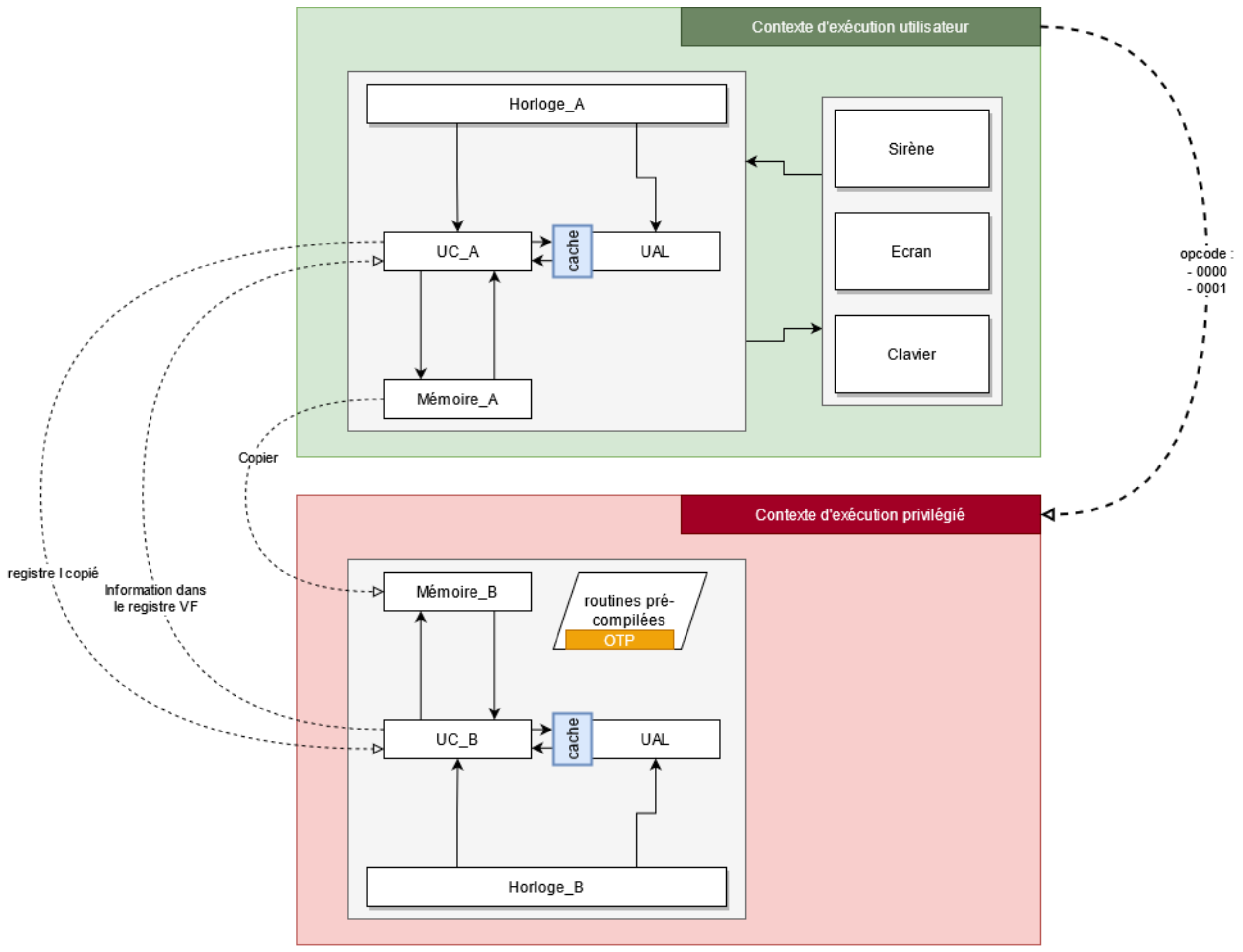
`CHIP-8` uses 4096 (0x1000) memory locations, all of which are **8** bits (a byte) which is where the term `CHIP-8` originated. However, the `CHIP-8` interpreter itself occupies the first **512** bytes of the memory space on these machines. For this reason, most programs written for the original system begin at memory location **512** (`0x200`) and do not access any of the memory below the location **512** (`0x200`). The uppermost **256** bytes (`0xF00-0xFFF`) are reserved for display refresh, and the **96** bytes below that (`0xEA0-0xEFF`) were reserved for the call stack, internal use, and other variables.

`CHIP-8` has **16 8-bit** data registers named `V0` to `VF`. The `VF` register doubles as a flag for some instructions; thus, it should be avoided. In an addition operation, `VF` is the carry flag, while in subtraction, it is the "no borrow" flag. In the draw instruction `VF` is set upon pixel collision.

The address register, which is named `I`, is **16 bits** wide and is used with several opcodes that involve memory operations.

The **stack** is only used to store return addresses when subroutines are called. The original RCA 1802 version allocated **48** bytes for up to **12** levels of nesting; modern implementations usually have more.

Chip-8 - FCSC Edition (translated with the help of DeepL for precision purposes)



Architecture of our Chip-8 emulator

The secure element is not attached to any device. It contains pre-compiled code in which the previously generated secret key is added.

Almost all operations take place in one cycle, except for operations that require the use of the arithmetic and logic unit (ALU), in which case two cycles are required: one cycle to submit the calculation to the ALU, and one cycle to retrieve the result. To overcome this performance problem, the manufacturer has developed a **cache system shared between the user VM and the secure element**. The cache is based on an LRU-type algorithm and is placed upstream of the ALU. Thus, if an operation has recently been performed, the CPU is directly aware of the result, hence costing only one cycle.

For the cache management, the constructor added the operation code `00E1`. This new instruction allows us to empty all the lines of the ALU cache.

The manufacturer has also implemented two other operation codes: `0000` and `0001`. They allow us to interact with the secure element by asking it to encrypt a 10-byte entry with the secret key or to ask it to check whether a 10-byte entry matches the secret key. The code for both routines is provided. A third operation code is implemented: `FFFF`. It is used to notify the VM of the end of the ROM execution.

Resolution

The instructions lead me to a number of questions :

- How is the secret generated ?
- How can I interact with the secret ?
- What are the result of my interactions with the secret ?

Luckily, the challenge provides us with the sources of the emulator, so let's dive into it to look for answers to our questions.

Reading the source code

I start by opening `challenge2.py`. It does not contain much, and leads us to the examination of `emulator.py`.
It contains the main code of the emulator. More specifically, it contains the initialization of the secret key (`execution_key`) as a **10-byte** random value. It then loads the `UntrustedContext` with the ROM data we provided, and runs it. As the FCSC specification stands, the ALU is not reset when changing context (the corresponding line is commented in `CPU.reset()`).
Let's then check the `cu.py`, the `ControlUnit` implementation file.
From the beginning, we see an interesting import : `import flag`. Following the references to `flag`, we land on this code snippet :

```
n = opcode & 0x0FFF
if msn == 0:
    if n == 0x0:
        se = self.cpu.context.secure_element
        se.reset()
        se.set_context(
            i=self.cpu.processor.i,
            memory=self.memory.data[self.memory.O_STACK:self.memory.SIZE]
        )
        se.exc_encrypt()
        self.cpu.processor.v[0xF] = se.cpu.tick
    elif n == 0x1:
        se = self.cpu.context.secure_element
        se.reset()
        se.set_context(
            i=self.cpu.processor.i,
            memory=self.memory.data[self.memory.O_STACK:self.memory.SIZE]
        )
        flag = se.exc_verify()
        self.cpu.processor.v[0xF] = se.cpu.processor.v[0xF]
        if self.cpu.processor.v[0xF] == 0:
            if self.cpu.processor.i+10+len(flag) > self.memory.SIZE:
                raise MemoryError("Not enought memory space")
            self.memory.data[self.cpu.processor.i+10:self.cpu.processor.i+10+len(flag)] = flag
    else:
        IllegalInstructionError(opcode)
```

So we can get the flag on the stack if `vF` equals **0** after calling the `exc_verify` routine. Also, we can observe that the `0000` instructions (to encrypt) will return their duration in number of ticks in `vF`.

Then, we have to understand what these macros are doing, in order to successfully make `exc_encrypt` return **0**.

- `exc_encrypt` :

```
preco_encrypt = """
6301
62{:02X}
F065
8023
F055
F31E

62{:02X}
F065
8023
F055
F31E

[...] (7 blocks similar to previous block)

62{:02X}
F065
8023
```

```
F055
F31E
FFFF
"""
```

This translates to :

```
v3 = 1
v2 = key[0]
v0 = mem[I]
XOR v0, v2
mem[I] = v0
I = I + v3

v2 = key[1]
v0 = mem[I]
XOR v0, v2
mem[I] = v0
I = I + v3

[...]

v2 = key[9]
v0 = mem[I]
XOR v0, v2
mem[I] = v0
I = I + v3
end program
```

which reverses to :

```
for i in range(10):
  v2 = key[i]
  v0 = mem[I]
  v0 = v0 ^ v2
  I += 1
```

In the same way, the `exc_verify` reverses to :

```
v2 = 0
for i in range(10):
  v3 = key[i]
  v0 = mem[I]
  v3 = v3 ^ v0
  v2 = v2 | v3
  I += 1
return v2
```

Given all that information, we can get a better view of our goal : make a call to `exc_verify` which must return `0`, meaning we must give the secret key as a parameter. When this is done, the flag will be inserted in memory and we will be able to read it the same way we read the secret in `Chip & Fish`.

Setting up a debug environment

To have an easier time crafting my ROMs, I chose to setup locally a debug environment. Thanks to the challenge authors giving us a working out-of-the-box copy of the emulator's code, setting that up only meant adding print instructions wherever I found a need for it. This included printing the content of the stack and the registers upon an `exc_encrypt` or `exc_verify` call, as well as printing cache misses with the parameters for XOR ALU operations. I also replaced the flag with the string `aaaaaaaaaa`, and got rid of the randomness of the secret key for an easier-to-track `ABCDEFGHIJ`.

```
elif opcode == 0xFFFF:
    print("Stack = ", self.memory.data[self.memory.O_STACK:self.memory.SIZE])
    print("I = ", hex(self.cpu.processor.i))
    print("Etat des registres :", self.cpu.processor.v)
    exit(0)
```

Example of a debug print statement, upon the instruction `ffff`

```
@request
@functools.lru_cache(maxsize=16)
def xor(self, a, b):
    print("Cache miss", a, b)
    return a ^ b, 0
```

Example of a debug print statement, upon `XOR` operations

Throughout the challenge, I used this page referencing the Chip-8 opcodes to help me with the programming.

We now know we have to leak the secret key in order to get the flag. Let's get to work !

First step : how do I leak the first byte ?

The `exc_encrypt` instruction takes the next 10 bytes starting from position `I` in memory, and `XORs` it with the private key. The only bytes we can control in the memory being on the stack, I am going to point `I` to the beginning of the stack. Remember the ALU, with the cache being shared between the trusted and untrusted context ? And the `exc_encrypt` returning its duration ? I have an intuition that we can get information about the bytes of the secret key based on how long it took to encrypt the flag. As we know, the `exc_encrypt` will perform XOR operations like `XOR mem[I + k], key[k]`. We control `mem[I + k]`, so we could set up the cache to contain a bunch of useless instructions as well as an instruction `XOR mem[I + k], K`. If `key[k]` isn't `K`, it will result in a cache miss. But if it is `K`, this will be a cache hit and the routine will take one less cycle to run. Let's check that, with debug printing the number of ticks it took to execute the `exc_encrypt` routine.

The algorithm is as follows :

1. Point `I` to the beginning of the stack
2. Call the `exc_encrypt` routine

This should give us the maximum numbers of cycles for the `exc_encrypt` routine, as the cache is empty and all the bytes of the secret key are different from one another.

> *For readability purposes, all my algorithm will be detailed in pseudo-code. They will always be easily translated to Chip-8 instructions. For instance, the above algorithm will translate to `aea0 0000 ffff`, where `aea0` corresponds to step 1, `0000` to step 2, and `ffff` means the end of the program.*

It took **61 ticks** to run the routine.

Now let's check with a correct XOR in the cache :

1. Point I to the beginning of the stack
2. Add the instruction XOR 0, 0x41 to the cache
3. Run the exc_encrypt routine

60 ticks ! One less, as expected

> *Keep in mind that these values depend on the fact that every byte of my key is different. If it wasn't the case, as the stack bytes are all the same (for the moment), we would have had a cache hit everytime we encountered a repeating byte.*

So, in a scenario where the first byte is unknown, we could retrieve it with the following algorithm (`A` , `B` and `C` represent any Chip-8 register) :

1. Point `I` on the stack
2. Set `A = 0` *(this will represent our hypothesis on the first byte)*
3. Set `B = 0` *(this will represent the value on the stack, at offset I + 0)*
4. Clear the cache *(for now, I just know I have to do this but I don't know how yet)*
5. `XOR B, A` *(to put this instruction in the cache)*
6. Call `exc_encrypt`
7. If `vF == 60` , store `A` in `C` *(if `vF == 60` , it means we had a cache hit, so our hypothesis is correct)*
8. If `A != 255` , `A = A + 1`
9. If `A != 0` , jump to step 3

Note : I implemented the "if - then" structure with "skip if …" instructions.

Step 7 should only match once, so at the end of this process `C` should contain the first byte of the key.

But this leaves me with a question : **how do I clear the cache ?**

If I had read correctly the FCSC description of the architecture, I could have used one instruction, `00E1` . But… I didn't.

So here is how I did it :
As a reminder, the cache works with LRU *(Least Recently Used)* policy and is of size **16**. So if I can call 16 irrelevant and unique `XOR` instructions, any previous data will be older than 16 and I will have filled the cache with useless data. All future accesses will thus result in cache misses. As all the relevant XORs are in the form `XOR mem[I + k], key[k]` and the cache does not recognize the commutative aspect of `XOR` , "irrelevant XOR instructions" just mean instructions in the form `XOR X, Y` where `X` is not present in memory between `I` and `I + 10` .

Thus the routine to clear the cache is as follows (with `X` and `Y` arbitrary Chip-8 registers) :

1. Set `Y = 0`
2. Set `X = ff`
3. `XOR X, Y`
4. `Y = Y + 1`
5. If `Y != 0x12` , jump to step 2 *(0x12 is slightly larger than needed, but I prefer it that way because now I am sure the 16 spots in the cache will be overwritten)*

While solving, I wasn't sure if I could store values on registers `v0` , `v1` , `v2` , and `v3` , as they were modified by the `exc_*` routines. Only much later in the process did I realize that registers were separate between contexts. So for now, I used `A = v4` , `B = v5` , `C = vE` , and `X = v1` , `Y = v2` as I did not care losing `X` and `Y` values between each call to an `exc_*` routine.

I implemented by hand the preceding algorithm, but it did not end-up as expected… **Why ?**
My debug information shows that, as every byte of the stack is equal, my cache entry will speed up the calculation at any index as long as the byte of the key is equal to A, and not only the first byte…

Solution : have a different value for the byte at index `I` in the stack, to make the `XOR` unique and have the speed-up only affect the wanted `XOR` .
How to set a byte on the stack ? Just call from the address you want, it will be stored on the stack.

Let's try again (I added a parameter `i` , which I will use later. For now, consider `i = 0`):

Algorithm 1 / *Parameters : int `i` , registers `A` , `B` , `C`*

1. Do a bunch of `nop` (I use instruction `6100` (`v0 <- 0`) as `nop`)
2. Call from an address where the two bytes differ, to any address (I jump to the next address for simplicity). This is done to ensure the unicity of the first byte in the stack.
3. Set `A = 0`
4. Point `I` on the first unique byte on the stack.

5. Set `B = mem[I + i]` *(as you know from which address you are jumping, you can hardcode `mem[I + i]`)*
6. Clear the cache
7. `XOR B, A` *to add it to the cache*
8. If `vF == 60`, store `A` in `C`
9. If `A != 255`, `A = A+1`
10. If `A != 255`, jump to step 4

And **it works** ! Looking at our debug output, we can see `C` (= `vE`) contains `0x41` at the end of our program !

Leak the second byte ?

The algorithm looks very much the same as the previous one, because jumping gave us two unique bytes on the stack. We just need to replace `mem[I]` with `mem[I + 1]`. `I` keeps pointing to the first unique byte on the stack, as we want `I + 1` to line up with the second byte. So we just have to run `Algorithm 1` with `i = 1`. And again, **this works**, according to our debug statements.

Now **how can we get both bytes in one execution ?**
From this point on, I started scripting my ROM generation. The algorithm borrows a lot of instructions from the preceding, and looks like that :

1. `Algorithm 1` with `i = 0`, `A = v4`, `B = v5`, `C = vD`
2. Steps 3 to 10 of `Algorithm 1` with `i = 1`, `A = v4`, `B = v5`, `C = vE`

Again, our debug output displays vD & vE at their expected values of 0x41 & 0x42. **Another step towards success !**

Leak the 10 bytes of the secret key in one go ?

Generalizing previous experiments, I proceeded as follows :

Algorithm 2 :

1. Setup the stack so that it contains **10** unique bytes
2. Re-use `Algorithm 1` with `i` going from **0** to **9**, changing register `C` each time.

But there I ran into a problem : I had not enough registers to run my algorithm, hold **10** key bytes, and leave `v0-v3` & `vF` free for the `exc_encrypt` routine. There, I realized empirically and then through reading the code again, that registers are not shared between contexts. So I can freely use `v0` or `v3` to hold key bytes (I already used `v1` & `v2` in my cache-clearing algorithm)

That being solved, **how do I setup the stack in such a way ?**
Easy : use jumps, to put the addresses on the stack. As we want **10** unique bytes, we will have to jump from **5** different, carefully chosen addresses (`0x0212`, `0x0314`, `0x0416`, `0x0518`, `0x061a`) so that the stack looks like : `0x021403160418051a061c`

It looks something like this :

```
0x200 : 6100
0x202 : 6100
[...]
0x20e : 6100
0x210 : 6100
0x212 : 2214
0x214 : 6100
[...]
0x312 : 6100
0x314 : 2316
0x316 : 6100
[...]
0x412 : 6100
0x414 : 6100
0x416 : 2418
0x418 : 6100
```

```
[........]
0x61a : 261c
```

Not very time-efficient, but **it works** and it is easy to write !

Now I have all **10** bytes of the secret key in **10** registers. I just have to re-order them and to store them on the stack (with the `FF55` instruction) to have the perfect setup for `exc_verify`.

Let's run that through our debug setup : after a couple tweaks and fixes, it works ! The stack contains the secret key. I retrieve its offset, to prepare for the next step : calling `exc_verify`.

This is simply done by adding 2 instructions at the end of Algorithm 2 : set I to the correct value (the value where our secret key begins), and call `exc_verify` (`0001`)

Debug prints show us that our mock flag **is** on the stack at the end of this program. **Success !**

Final step : displaying the flag

Let's grab the flag offset on the stack with our debug infos, and use the same technique as I used in `Chip & Fish` :

Algorithm 3 / *Parameters : int `addr` , `n` , registers `A` , `B`*

1. Point `I` at `addr`
2. Set `A = 0`
3. Set `B = 0`
4. Draw the sprite at address `I` , position `x = A` , `y = B` (instruction `DAB1`)
5. If `B != n` , `I = I + 1`
6. If `B != n` , `B = B + 1`
7. If `B != n+1` , jump to step 4

The final payload goes :

1. `Algorithm 2`
2. `Algorithm 3` , `addr` being the address of the flag we got through debug prints, `n` being the number of bytes we want to read (it turns out the flag is **16** bytes long), and `A` & `B` two arbitrary registers.

After a quick binary to hexadecimal conversion, **we get the flag !**

> *In reality, I used a slightly different method to retrieve the secret key. Once the stack is initialized, instead of increasing the register B in Algorithm 1, I decreased the register I and kept B constant. This solution is worse than the one I presented, because at round k, the last 10-k bytes of the stack are all 0, and if two bytes of the key are the same, let's say K, the operation XOR 0, K will occur twice in the exc_encrypt routine, and the timings will be broken in my program. But this occurs with probability < 0.5, so in the case it doesn't work on the first try, I could just re-run my script once or twice.*

Final words

I really enjoyed this series of challenges (`Chip & Fish` + `BattleChip`). They were very well made challenges, and the emulator's code was nicely readable. A big thank you to the ANSSI team for setting up challenges like these.