OS-S_V1.17

> Ce challenge est un challenge à tiroir, à chaque étape vous pourrez valider un flag sous la forme DGA{XXX} dans l'étape correspondante.
>
> Vous devez passer le Tset pour acceder à cette machine
>
> Url: tcp://ossv117.chall.malicecyber.com:4993/

We get the following bash script to launch the disk (`os-image.bin`) that holds the flags as well as the disk.

```
#!/bin/bash

qemu-system-x86_64 \
    -snapshot \
    -drive format=raw,readonly,index=0,if=floppy,file=os-image.bin \
    -hdb flag_04.txt \
    -serial stdio
```

The program running on the server is the following:

```
if __name__ == '__main__':
    signal.signal(signal.SIGALRM, sig_handler)
    signal.alarm(120)

    proof_of_work()

    choise = input("Voulez-vous utiliser une configuration personnalisée pour le
processeur ? [O/N] : ")

    if choise == "O":
        cpu = str(input("Configuration cpu: "))

        import string
        if not all([c in string.ascii_letters + string.digits + "," + "=" + '-' +
'.' for c in cpu]):
            print("Caractères non autorisés.")
            exit()

    else:
        cpu = "Westmere"

    argv = ["/usr/bin/timeout", "120",
            "/usr/bin/qemu-system-x86_64",
            "-snapshot",
            "-display", "none",
            "-serial", "stdio",
            "-drive", "format=raw,readonly,index=0,if=floppy,file=os-image.bin",
```

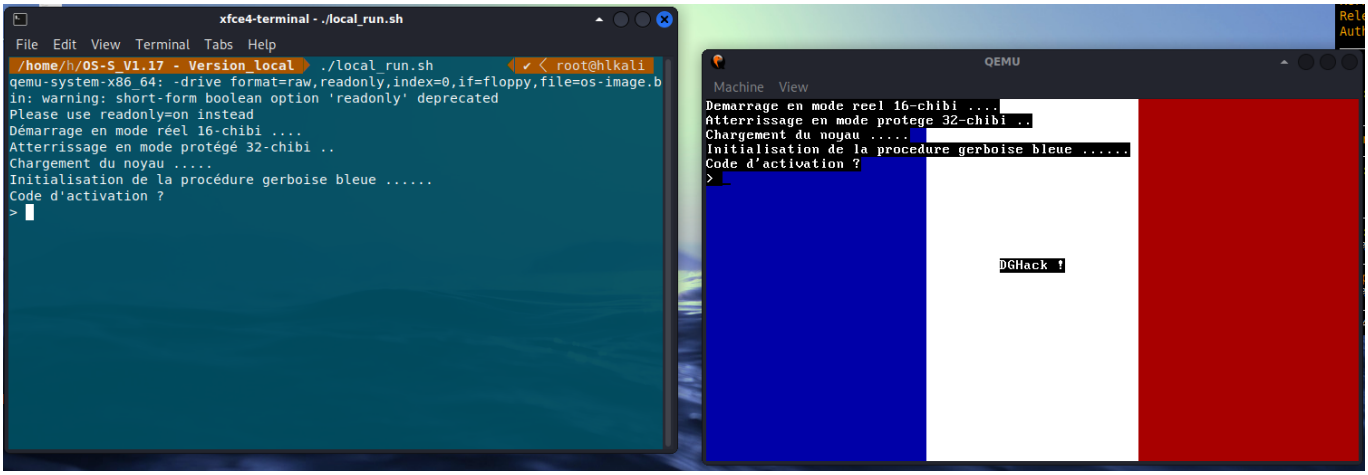
```
            "-hdb", "flag_04.txt",
            "-cpu", cpu
            ]

    env = {}
    os.execve(argv[0], argv, env)
```

(the `sighandler` and `proof_of_work` functions are omitted for clarity).

Description

The OS image seems to be a custom-made OS that we can launch with QEMU. We can also choose a CPU.

This is what I get when I launch the program.



I first tried to reverse the program statically but could not achieve it.

From my understanding, the program starts in 16-bit mode real, then takes the code and copies it on another part of the memory, and runs it in 32-bit protected mode.

To debug the program with gdb, launch the QEMU system with `-s -S` as additional parameters. On another console, launch gdb and type `target remote localhost:1234` to connect to the process.

The actual start of the program will be at address 0x7c00 so I set a breakpoint there, and then advanced in the program using breakpoints (`b * address`) and continue (`c`).

To disassemble code, I'm using `x/20i address` or `x/20i $pc` to disassemble at the current address (you can replace 20 instructions to whatever value you want).

Going through the code functions by functions, waiting for messages to appear and trying to find the spot where I need to give my input, I understood the following.

```
0x1552: call   0x12e1
0x1557: call   0x135f
0x155c: call   0x1448
0x1561: call   0x13d1
0x1566: call   0x150f
0x156b: jmp    0x1566
```

At address 0x1552, there is a call to 5 functions.

- I don't know what the first function does but it does not appear to be relevant.
- Function at 0x135f shows the first 3 lines of the first screen.
- Function at 0x1448 is for the first challenge.
- Function at 0x13d1 is for the second challenge.
- Function at 0x150f is for the third challenge (probably).

Let's now look at each challenge one by one.

Part one

Function 0x1448 has the following code:

```
; [REDACTED, it just prints the remaining lines]
0x1492: push   0x50
0x1494: push   rsi
0x1495: call   0x1ce0
0x149a: pop    rcx
0x149b: pop    rbx
0x149c: lea    ebx,[rbp-0x58]
0x149f: push   rbx
0x14a0: push   rsi
0x14a1: mov    BYTE PTR [rbp-0x59],0x0
0x14a5: call   0x1029
0x14aa: add    esp,0xc
0x14ad: push   0x4f
0x14af: push   0x2be0
0x14b4: push   rbx
0x14b5: call   0x257c
0x14ba: add    esp,0x10
0x14bd: test   eax,eax
0x14bf: je     0x14d3
```

There are 3 function calls:

- function 0x1ce0 reads the input,
- function 0x1029 transforms the input,
- function 0x257c compares the transformed input with some stored data.

If the password is not correct, the jump is not taken and we get an error message and the program stops. I simulated to get the correct message by jumping to 0x14d3 (by typing `jump * 0x14d3` when I was on the `je` instruction), and the flag appears (a fake flag of course, the real one is on the remote).

So I need to understand how the string is transformed and then to reverse the function.

Here I have annotated the assembly, and put a summary of it for better understanding.

```
; First the function prequel
0x1029: push   rbp
0x102a: mov    ebp,esp
0x102c: push   rdi
0x102d: push   rsi
0x102e: push   rbx

; local variables definitions
0x102f: xor    ebx,ebx                      ; i = 0
0x1031: sub    esp,0x28
0x1034: mov    esi,DWORD PTR [rbp+0x8]      ; my_input
0x1037: mov    edi,DWORD PTR [rbp+0xc]      ; other
; other is an array given as parameter
; its initial content are actually unimportant

; rbp-0x1c = strlen(my_input)
0x103a: push   rsi
0x103b: call   0x236f
0x1040: add    esp,0x10
0x1043: mov    DWORD PTR [rbp-0x1c],eax

; main loop
; while i < strlen(my_input)
0x1046: cmp    ebx,DWORD PTR [rbp-0x1c]
0x1049: jg     0x10a2

    ; eax = i//3 and edx = i%3
    0x104b: mov    eax,ebx
    0x104d: mov    ecx,0x3
    0x1052: cdq
    0x1053: idiv   ecx

    ; al = my_input[i]
    0x1055: mov    al,BYTE PTR [rsi+rbx*1]

    ; ecx = &other[i]
    0x1058: lea    ecx,[rdi+rbx*1]

    ; comparisons on i%3 that lead to the different ifs
    0x105b: cmp    edx,0x1
    0x105e: je     0x1072
    0x1060: cmp    edx,0x2
    0x1063: je     0x1087

    ; if i%3 == 0
        0x1065: xor    eax,0x42
        0x1068: mov    BYTE PTR [rdi+rbx*1],al ; other[i] = my_input[i] ^ 0x42
        0x106b: push   rax
        0x106c: push   rax
        0x106d: movsx  eax,bl                  ; eax = i
        0x1070: jmp    0x1095
    ; fi => jump to 0x1095
```

```
    ; elif i%3 == 1
        0x1072: xor    al,BYTE PTR [rsi+rbx*1-0x1]
        0x1076: mov    BYTE PTR [rdi+rbx*1],al ; other[i] = my_input[i] ^
my_input[i-1]
        0x1079: movsx  eax,bl
        0x107c: push   rdx
        0x107d: push   rdx
        0x107e: push   rax
        0x107f: push   rcx
        0x1080: call   0x1018 ; ror(other[i], i)
        0x1085: jmp    0x109c
    ; fi => jump to 0x109c

    ; elif i%3 == 2
        0x1087: xor    al,BYTE PTR [rsi+rbx*1-0x1]
        0x108b: mov    BYTE PTR [rdi+rbx*1],al ; other[i] = my_input[i] ^
my_input[i-1]
        0x108e: push   rax
        0x108f: push   rax
        0x1090: movsx  eax,BYTE PTR [rsi+rbx*1-0x1] ; eax = my_input[i-1]
    ; fi => just continue

    ; this part happens when i%3 == 0 or 2
    0x1095: push   rax
    0x1096: push   rcx
    0x1097: call   0x1007             ; rol(other[i], eax)

    ; end of loop: add 1 to i and return above
    0x109c: add    esp,0x10
    0x109f: rex.XB jmp 0x1046
; end of loop

; function end
0x10a2: lea    esp,[rbp-0xc]
0x10a5: pop    rbx
0x10a6: pop    rsi
0x10a7: pop    rdi
0x10a8: pop    rbp
0x10a9: ret
```

To summarize, if I were to translate this function to pseudocode, I would get:

```
for i in range(len(my_input)):
    if i%3 == 0:
        other[i] = my_input[i] ^ 0x42
        other[i] = rol(other[i], i)
    elif i%3 == 1:
        other[i] = my_input[i] ^ my_input[i-1]
        other[i] = ror(other[i], i)
    else:
        other[i] = my_input[i] ^ my_input[i-1]
        other[i] = rol(other[i], my_input[i-1])
```

where `rol` and `ror` are the standard functions to rotate bits (on 8 bits integers).

From the next function, I also get the array the transformed string (called `other` here) is compared to.

Reversing it is easy: I can construct the original string from the beginning character by character. Here is the solving script:

```
final = [0x14, 0x9c, 0xd, 0x89, 0x35, 0x53, 0xc8, 0x2e, 0xc4, 0x70, 0x95, 0xc0,
0xf2, 0x10, 0x3, 0x31, 0x4a, 0x3c, 0x89, 0xc9, 0xc2, 0x4c, 0xd, 0x88, 0x2d, 0x3,
0x34, 0x13, 0x5, 0x11, 0x4c, 0xa6, 0x51, 0x6e, 0x94, 0x27, 0xe2, 0x62, 0x59, 0x31,
0x41, 0x10, 0xd8, 0x8a, 0x44, 0xe4, 0x15, 0x42, 0x2d, 0x80, 0x93, 0x69, 0xa1,
0xaa, 0x89, 0x2, 0xa8, 0x5e, 0x3, 0x28, 0x43, 0xb8, 0x10, 0x98, 0x53, 0x4d, 0xb4,
0x63, 0x45, 0xc4, 0x4, 0xa8, 0x32, 0x88, 0x24, 0x89, 0x61, 0x69, 0x90]

retrieved_input = []

# Honteusement copié depuis
https://gist.github.com/trietptm/5cd60ed6add5adad6a34098ce255949a
# Rotate left: 0b1001 --> 0b0011
rol = lambda val, r_bits, max_bits: \
    (val << r_bits%max_bits) & (2**max_bits-1) | \
    ((val & (2**max_bits-1)) >> (max_bits-(r_bits%max_bits)))

# Rotate right: 0b1001 --> 0b1100
ror = lambda val, r_bits, max_bits: \
    ((val & (2**max_bits-1)) >> r_bits%max_bits) | \
    (val << (max_bits-(r_bits%max_bits)) & (2**max_bits-1))

max_bits = 8

for i in range(len(final)):
    if i%3 == 0:
        v = ror(final[i], i, max_bits)
        retrieved_input.append(v ^ 0x42)
    elif i%3 == 1:
        v = rol(final[i], i, max_bits)
        retrieved_input.append(v ^ retrieved_input[i-1])
    elif i%3 == 2:
        v = ror(final[i], retrieved_input[i-1], max_bits)
        retrieved_input.append(v ^ retrieved_input[i-1])

print(bytes(retrieved_input))
```

The password: `Vous savez, moi je ne crois pas qu'il y ait de bon ou de mauvais mot de passe.`

Flag: `DGA{R0tat1on_d3_ch1bis}`