Problem 2 : Reverse Engineering

1. (2 points) Disassemble the instruction at 0x8049197, which loads a value onto the stack. What is the value ? In which section is the value stored ?

The instruction at 0x8049197 is « push dword ptr [0x804c028] »

So first of all *push* means to place it on the stack.

dword ptr, we are looking at an address which is pointing to a dword which is 4 bytes. The address that contains the dword is [0x804c28] and we are pushing the 32 bits value that is in the memory location 0x804c028.

When we are doing our b2r2 dump serial, we can see in the output a (.data) section, with the address 0x804c020, we are pushing the value at the memory location 0x804c028.

```
# (.data)

0804c020: 00 00 00 00 00 00 00 00  08 A0 04 08           | ........*.**
```

The value is stored in the .data section and the value is : 8

To find it I used gdb and used the command x/d 0x804c028

```
(gdb) x/d 0x804c028
0x804c028 <serial>:     8
```

2. (2 points) Disassemble the instructions at 0x80491a2, which calls the strcpy function. What parameters are passed to the function ?

The instruction at 0x80491a2 is « call -0x162 ; <strcpy> »

Strcpy is a function in C : *char *strcpy(char *dest, const char *src)* it copy the string pointed to, by src to dest.

I am using gdb and I am setting up a breakpoint at the main function, then I use ni to step through it.

```
0x0804919d <+11>:    lea    eax,[esp+0x6]
0x080491a1 <+15>:    push   eax
0x080491a2 <+16>:    call   0x8049040 <strcpy@plt>
```

If you look at the instruction, you have got *lea eax, [esp+0x6]* which means that eax will contains the address of [esp+0x6] and at this address you have got a value.

So before it is executed, with info registers you can see the value of eax.

```
eax            0xf7fc1ae8          -134472984
```

And at the next instruction, you have got

```
eax            0xffffd5b2          -10830
```

You can also see that it is indeed, the value of esp+0x6

```
esp            0xffffd5ac          0xffffd5ac
```

Then you push eax, it means you are putting the value in eax, in the stack and I think it is this value that you send to the strcpy function.

```
(gdb) x/d 0xffffd5b2
0xffffd5b2:     -5
```

As we saw, strcpy get 2 parameters (you can see it if you execute man strcpy), also it says the return value of the function is a pointer to the destination string dest.

```
STRCPY(3)                                   Linux Programmer's Manual

NAME
       strcpy, strncpy - copy a string

SYNOPSIS
       #include <string.h>

       char *strcpy(char *dest, const char *src);
```

So the push before was the one from the previous question, so we are giving to our strcpy function, the parameters (-5 and 8), or actually the string at address 0x804a008 to the destination.

```
(gdb) x/a 0x804c028
0x804c028 <serial>:     0x804a008
(gdb) x/s 0x804a008
0x804a008:      "E6X7B$0D1BY<6UxXA(E6Cc5VU:5Z1]jK"
```

After that, I did a few ni, and then

```
0x080491ad in main ()
(gdb) x/s 0xffffd5b2
0xffffd5b2:     "E6X7B$0D1BY<6UxXA(E6Cc5VU:5Z1]jK"
```

The string was copied at address 0xffffd5b2

3. (2 points) Dissasemble the instruction at 0x80491bb. This conditional branch constructs a loop. What kind of condition is checked with this instruction ?

The instruction at 0x80491bb is « jg +0x26 ; 0x80491e1 »

Just on top of this, we have a line that is *cmp eax, 0x1f*, so we are comparing the value of eax to 1f and if eax is greater than 0x1f which is 31 in decimal, then we are going to the instructions at the memory address 0x80491e1.

So actually before our instructions at 0x80491bb, we have a bunch of instructions, but what is really important to us is actually what is in eax when the command *cmp eax, 0x1f* is executed so let's use gdb and set a breakpoint at 0x80491bb and use info registers to see what is in eax.

```
Breakpoint 3, 0x080491bb in main ()
(gdb) info registers
eax            0x1                 1
```

So we are comparing 0x1 and 0x1f when we are not giving any input to our program.

I did some test cases where I gave some input to the program because there is a read instruction at address 0x80491d2.

For example for a test file, where wc give us that, I got :

```
(gdb) shell wc test
 1  1 13 test
```

And in info registers :

```
Breakpoint 2, 0x080491bb in main ()
(gdb) info registers
eax            0xd                 13
```

So it looks like in eax, you have got the length of the string in the file you are giving as input.

Since 0x1f is equal to 31, I tried to put a string longer than 32 in the test file.

```
(gdb) shell wc test
 1  1 37 test
```

But no matter how long was the string, eax was always at value 32.

```
Breakpoint 2, 0x080491bb in main ()
(gdb) info registers
eax            0x20                32
```

So to pass this condition, the lenght of the string you are giving in input must be superior than 31 characters, so eax will be superior than 0x1f and then you will jump to address 0x80491e1

4. (2 points) Dissasemble the instruction at 0x80491fc. In which condition the call instructions is invoked ? What's the purpose of having this call ?

The instruction at 0x80491fc is « call -0x19c ; <exit> »

When you see the man page, the goal of the exit function is to :

```
EXIT(3)                                Linux Programmer's Manual                                EXIT(3)

NAME
       exit - cause normal process termination

SYNOPSIS
       #include <stdlib.h>

       void exit(int status);

DESCRIPTION
       The exit() function causes normal process termination and the value of status & 0xFF is returned to the parent
       (see wait(2)).
```

The purpose of having this call is to terminate the process if some of conditions are not met. But there are some in particular that should be checked. The one where you have jmp to 0x80491fa.

```
0x080491ed <+91>:    cmp    al,BYTE PTR [esp+0x23]
0x080491f1 <+95>:    jne    0x80491fa <main+104>
0x080491f3 <+97>:    mov    eax,0x0
0x080491f8 <+102>:   jmp    0x8049203 <main+113>
0x080491fa <+104>:   push   0x1
0x080491fc <+106>:   call   0x8049060 <exit@plt>
```

So the thing is that there are more cmp and jump before the instruction at 0x080491fc but the instructions at line 0x80491ed and 0x80491f1 are particularly important as you can see, the *cmp al, BYTE PTR [esp+0x23]*, if al and the value at [esp+0x23] are not equal we will jump to address 0x80491fa and then 0x80491fc that will exit the program.

If the conditions was verified, there would be no issue because at address 0x80491f8, you have a jump with no condition to an instruction at an address lower than our exit call.

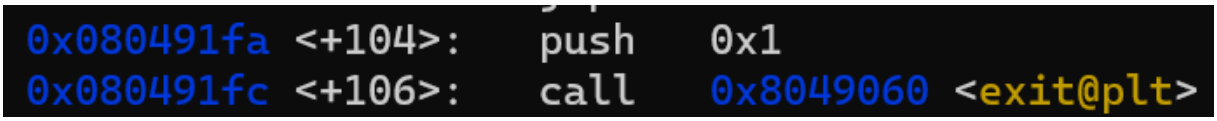
This means that al and the value at BYTE PTR [esp+0x23] should be strictly equal or the program will exit.

What is AL ? EAX is the 32 bits version, AX is the 16 lower bits of AX, then you have got AH which is the 8 high bits of AX and AL the least significant byte of EAX.
But when I try to use GDB to see which value were stored in AL and at the memory location [esp+0x23], I could not quite do it, because when i put a breakpoint at 0x80491f1, and I press *c* to continue the program until the next breakpoint I could have waited forever since it was like an infinite loop when I manually used *ni* or *si* to run the program.

There is another instructions that jump to 0x80491fa, at 0x804210, you have cmp byte ptr [ESP+EAX+0x023], CL and then at 0x8049214, jnz -0x1a (0x80491fa).

```
0x08049210 <+126>:   cmp    BYTE PTR [esp+eax*1+0x23],cl
0x08049214 <+130>:   jne    0x80491fa <main+104>
```

So the byte value at [esp+eax*1+0x23] and cl need to be equal to 0, else the program will trigger the exit call and stop the program with exit code 1.



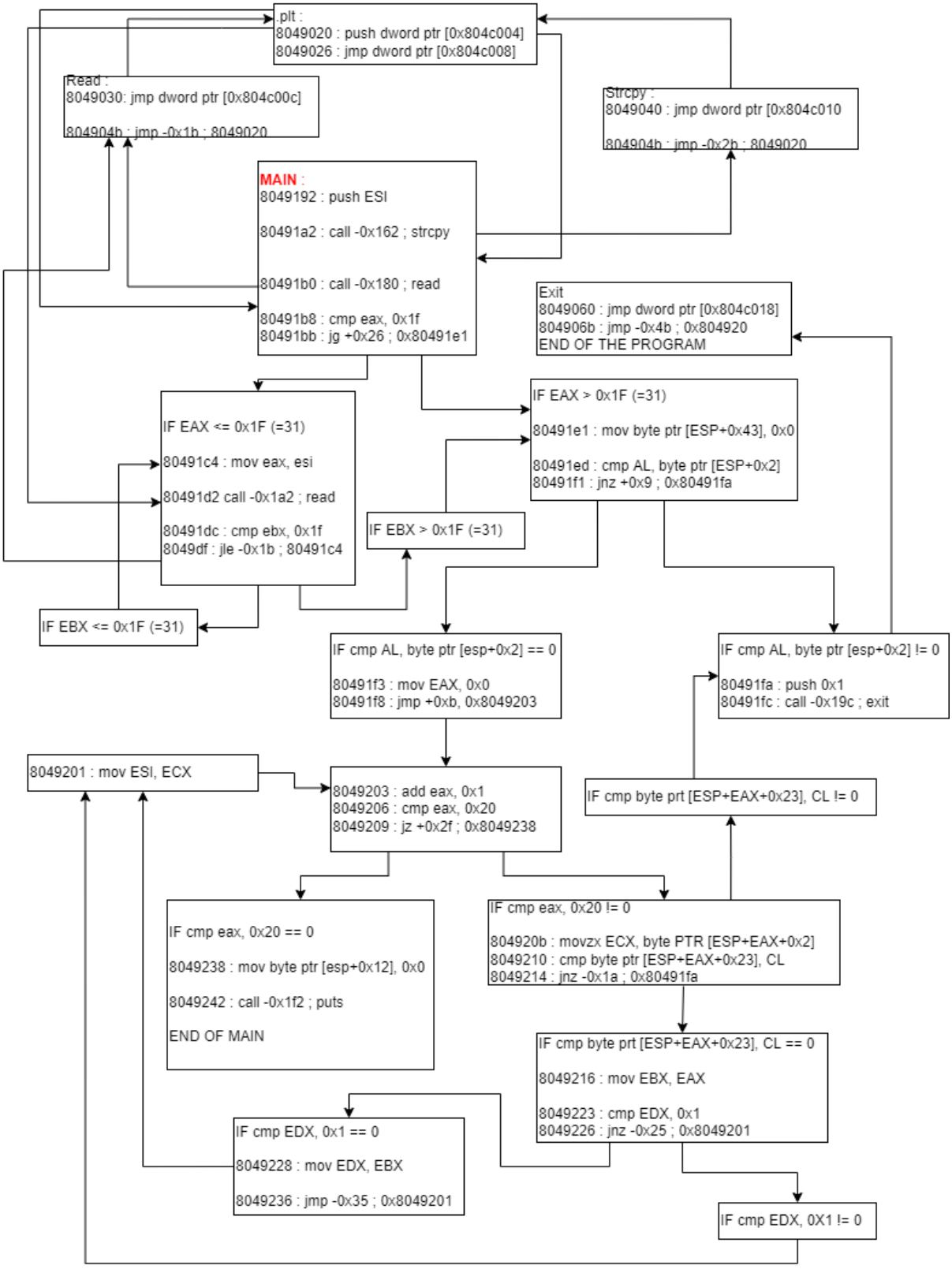
Indeed, the exit code would be 1, since we are pushing 0x1 in the stack just before we call the exit instruction.

But to get here, there are a lot of cmp then jump instructions. So there are quite a few things that are being checked throughout the program.

5. (10 points) Fully reverse engineer the main function and show the corresponding C code.

When I run the main, I am stuck in an infinite loop.

To reverse engineer it, I will first break the main function into blocks and try to get a better understanding of the control flow.


.plt :
8049020 : push dword ptr [0x804c004]
8049026 : jmp dword ptr [0x804c008]
Read :
8049030: jmp dword ptr [0x804c00c]
804904b : jmp -0x1b ; 8049020
Strcpy :
8049040 : jmp dword ptr [0x804c010
804904b : jmp -0x2b ; 8049020
MAIN :
8049192 : push ESI
80491a2 : call -0x162 ; strcpy
80491b0 : call -0x180 ; read
80491b8 : cmp eax, 0x1f
80491bb : jg +0x26 ; 0x80491e1
Exit
8049060 : jmp dword ptr [0x804c018]
804906b : jmp -0x4b ; 0x804920
END OF THE PROGRAM
IF EAX <= 0x1F (=31)
80491c4 : mov eax, esi
80491d2 call -0x1a2 ; read
80491dc : cmp ebx, 0x1f
8049df : jle -0x1b ; 80491c4
IF EAX > 0x1F (=31)
80491e1 : mov byte ptr [ESP+0x43], 0x0
80491ed : cmp AL, byte ptr [ESP+0x2]
80491f1 : jnz +0x9 ; 0x80491fa
IF EBX > 0x1F (=31)
IF EBX <= 0x1F (=31)
IF cmp AL, byte ptr [esp+0x2] == 0
80491f3 : mov EAX, 0x0
80491f8 : jmp +0xb, 0x8049203
IF cmp AL, byte ptr [esp+0x2] != 0
80491fa : push 0x1
80491fc : call -0x19c ; exit
8049201 : mov ESI, ECX
8049203 : add eax, 0x1
8049206 : cmp eax, 0x20
8049209 : jz +0x2f ; 0x8049238
IF cmp byte prt [ESP+EAX+0x23], CL != 0
IF cmp eax, 0x20 == 0
8049238 : mov byte ptr [esp+0x12], 0x0
8049242 : call -0x1f2 ; puts
END OF MAIN
IF cmp eax, 0x20 != 0
804920b : movzx ECX, byte PTR [ESP+EAX+0x2]
8049210 : cmp byte ptr [ESP+EAX+0x23], CL
8049214 : jnz -0x1a ; 0x80491fa
IF cmp byte prt [ESP+EAX+0x23], CL == 0
8049216 : mov EBX, EAX
8049223 : cmp EDX, 0x1
8049226 : jnz -0x25 ; 0x8049201
IF cmp EDX, 0x1 == 0
8049228 : mov EDX, EBX
8049236 : jmp -0x35 ; 0x8049201
IF cmp EDX, 0X1 != 0

So I will try to construct the corresponding C code with the control flow file I made. I mostly rearranged the code in IF and loop in order to better understand the assembly control flow.

You will find it below :

```c
// KAU Anthony - 20226189

// Just tried to put the different instructions in assembly
// in loop and with the condition
// Never really coded in C during my school time so had a bit
// of trouble trying to understand that

#include <stdio.h>
#include <string.h> // strcpy is in string

void main()
{
    // We have to get the string in the file that is given in input
    FILE* ptr;
    char str[50];
    ptr = fopen("File.txt", "r");

    if (NULL == ptr){
        printf("file can't be opened \n");
    }

    fgets(str, 50, ptr);


    char str1[] = "E6X7B$0D1BY<6UxXA(E6Cc5VU:5Z1]jK";
    char str2[];

    strcpy(str2, str1); // we are copying str1 in str2

    if (strlen(str) < 0x1f); // cmp eax, 0x1f, with question 3, we saw that
EAX contains the length of the string in the file given in input
    {
        EBX = EAX;
        ESI = 0x20;
        while (EBX < 0x1f) // cmp EBX, 0x1f
        {
            EAX = ESI;
            EAX = EAX - EBX;
            push EAX; // pushing EAX in the stack
            EAX = dword ptr [ESP+0x27]; // lea EAX, dword ptr [ESP+0x27], EAX
now holds an address
            EAX = EAX + EBX;
            push EAX; // pushing EAX in the stack
            push 0x0; // pushing 0 on the stack
```

```
            read();
            EBX = EBX + EAX;
            ESP = ESP + 0xc;
        }
    }

    mov byte ptr [ESP+0x43], 0x0;
    movzx ESI, byte ptr [ESP+0x23];
    EAX = ESI;

    if (AL != 0) // cmp AL, byte ptr [ESP+0x2]
    {
        exit (1);
    }

    EAX = 0x0;


    EAX = EAX + 1;

    While (EAX == 0x20) // cmp eax, 0x20
    {
        movzx ECX, byte ptr [ESP+EAX+0x2];

        if (byte ptr [ESP+EAX+0x23] != CL) // cmp byte ptr [ESP+EAX+0x23], CL
        {
            exit(1);
        }

        EBX = EAX
        shr EBX, 0x1f
        lea EDX, dword ptr [EAX+EBX]
        and EDX, 0x1;
        EDX = EDX - EBX

        if (EDX == 0x1) // cmp EDX, EBX
        {
            EDX = EBX; // mov EDX, EBX
            EDX = EDX + EAX;
            sar EDX, 0x1;
            EBX = ESI;
            mov byte ptr [ESP+EDX+0x2], BL
        }

        ESI = ECX; // mov ESI, ECX
    }

    mov byte ptr [ESP+0x12], 0x0;
```

```
    EAX = dword ptr [ESP+0x2]; // lea instructions, so EAX = Address
    push EAX; // push EAX in the stack
    puts(); // put display the string in EAX
    break; // end the for loop
    // END OF THE PROGRAM


    return 0;
}
```

6. (2 points) Which input do you need to provide to the program in order to observe the correct output at 0x8049242 ? What is the output ?

The input you need to provide to get a correct output at 0x8049242 is : the string « E6X7B$0D1BY<6UxXA(E6Cc5VU:5Z1]jK » which is in the first file from the screen below.

And you get as an output, the string « softsec is cool ! »

I found it by luck to be honest, so I don't know if I should get the points. It was like the only things that I could get from the serial file so I just put it as an input and it did work.

```
vagrant@cs492e:~/problem2$ ./serial < first
softsec is cool!
```

Problem 3 : Custom Debugger

1. (5 points) The secret stored in the binary is easily revealed with such a custom debugger. Patch the binary in such a way that the secret is not revealed anymore even with the custom debugguer that you made. Specifically, you should change the four instructions located at 0x80491e6, 0x80491ed, 0x804920b and 0x8049210 to achieve our goal. Explain.