Open the binary in GDB, set intel as the preferred asm syntax (if that's what you prefer), set breakpoint to main and run the binary.

```
$ gdb ./level2
(gdb) set disassembly-flavor intel
(gdb) b main
(gdb) run
```

You will hit the breakpoint we set earlier. Let's take a look at the disassembly. We will see a lot of stuff, but pay attention to the function names that are being called:

```
(gdb) disassemble
   <SNIP>
   0x000000000040077a <+47>:    call   0x4005b0 <memset@plt>
   <SNIP>
   0x0000000000400789 <+62>:    call   0x4005a0 <printf@plt>
   <SNIP>
   0x000000000040079f <+84>:    call   0x4005e0 <__isoc99_scanf@plt>
   <SNIP>
   0x00000000004007ab <+96>:    call   0x4006dd <check_password>
```

For our purposes, that `check_password` looks interesting. Let's set a breakpoint there and look at the disassembly. You will prompted for a password before hitting the breakpoint, type anything for now

```
(gdb) b check_password
Breakpoint 2 at 0x4006e1
(gdb) c
Continuing.
Password: anythingfornow

Breakpoint 2, 0x00000000004006e1 in check_password ()
(gdb) disassemble
Dump of assembler code for function check_password:
   0x00000000004006dd <+0>:     push   rbp
   0x00000000004006de <+1>:     mov    rbp,rsp
=> 0x00000000004006e1 <+4>:     sub    rsp,0x20
   0x00000000004006e5 <+8>:     mov    QWORD PTR [rbp-0x18],rdi
   0x00000000004006e9 <+12>:    mov    rax,QWORD PTR [rbp-0x18]
   0x00000000004006ed <+16>:    mov    rdi,rax
   0x00000000004006f0 <+19>:    call   0x400580 <strlen@plt>
   0x00000000004006f5 <+24>:    mov    DWORD PTR [rbp-0x4],eax
   0x00000000004006f8 <+27>:    mov    DWORD PTR [rbp-0x8],0x0
   0x00000000004006ff <+34>:    jmp    0x40073c <check_password+95>
   0x0000000000400701 <+36>:    mov    eax,DWORD PTR [rbp-0x8]
   0x0000000000400704 <+39>:    movsxd rdx,eax
   0x0000000000400707 <+42>:    mov    rax,QWORD PTR [rbp-0x18]
```

```
   0x000000000040070b <+46>:    add    rax,rdx
   0x000000000040070e <+49>:    movzx  eax,BYTE PTR [rax]
   0x0000000000400711 <+52>:    movsx  eax,al
   0x0000000000400714 <+55>:    mov    rcx,QWORD PTR [rip+0x200965]        # 0x601080
 <maybe>
   0x000000000040071b <+62>:    mov    edx,DWORD PTR [rbp-0x8]
   0x000000000040071e <+65>:    movsxd rdx,edx
   0x0000000000400721 <+68>:    add    rdx,rcx
   0x0000000000400724 <+71>:    movzx  edx,BYTE PTR [rdx]
   0x0000000000400727 <+74>:    movsx  edx,dl
   0x000000000040072a <+77>:    sub    edx,0x1
   0x000000000040072d <+80>:    cmp    eax,edx
   0x000000000040072f <+82>:    je     0x400738 <check_password+91>
   0x0000000000400731 <+84>:    mov    eax,0x0
   0x0000000000400736 <+89>:    jmp    0x400749 <check_password+108>
   0x0000000000400738 <+91>:    add    DWORD PTR [rbp-0x8],0x1
   0x000000000040073c <+95>:    mov    eax,DWORD PTR [rbp-0x8]
   0x000000000040073f <+98>:    cmp    eax,DWORD PTR [rbp-0x4]
   0x0000000000400742 <+101>:   jl     0x400701 <check_password+36>
   0x0000000000400744 <+103>:   mov    eax,0x1
   0x0000000000400749 <+108>:   leave
   0x000000000040074a <+109>:   ret
End of assembler dump.
```

There's quite a lot of stuff going on, but let's break it down to pieces:

- instructions from +0 to +16 set up the stack and the function parameters
- instructions from +19 to +27 initialize values on the stack (`strlen` is called and it's return value is stored to `rbp-0x4`). `rbp-0x8` is set to 0. So now we have two variables (both are 4 bytes long) on the stack, the length of something and something that is set to 0.
- +34 jumps straight to +95, where the value from `rbp-0x8` is compared to the value of `rbp-0x4`. If `rbp-0x8` is less than `rbp-0x4` the execution jumps back to +36. If they are are equal, eax is set to `0x1` and the function returns. So it looks like `rbp-0x8` is a counter in a for loop and `rbp-0x4` is the maximum value of the counter. If you inspect the value of `rbp-0x4` you will notice that it matches the length of the password you typed in earlier.
- +36 moves the counter value to `eax` then +39 moves it to `rdx`.
- +42 and +46 move `rbp-0x18` to rax and add `rdx` to `rax`.
- Instruction from +49 to +68 move a single byte to `eax` and add `rbp-0x8` to `rip+0x200965`
- +71 and +74 retrieves a byte from the address that resulted from the calculation in the previous step.
- +77 substracts 0x1 from the byte
- +80 to +89 compare eax to edx and if they're equal, continue another round of the for loop. Otherwise the function exits with 0x0.

In other words, something is looped over and on each loop 1 is substracted from a value, which is then compared to

another value. If the values match the loop continues, if not the whole function returns with 0x0 (false) if the all the values match and the loop terminates, the function returns with a value of 0x0 (true).

From this we can conclude that the password is stored in memory in a scrambled format and it can unscrambled by substracting 1 from each character.

Let's try to locate the password from memory by setting a breakpoint right before the comparison instruction:

```
(gdb) b *0x000000000040072a
(gdb) c
Breakpoint 3, 0x000000000040072a in check_password ()
(gdb) disassemble
<SNIP>
   0x000000000040071b <+62>:    mov    edx,DWORD PTR [rbp-0x8]
   0x000000000040071e <+65>:    movsxd rdx,edx
   0x0000000000400721 <+68>:    add    rdx,rcx
   0x0000000000400724 <+71>:    movzx  edx,BYTE PTR [rdx]
   0x0000000000400727 <+74>:    movsx  edx,dl
=> 0x000000000040072a <+77>:    sub    edx,0x1
   0x000000000040072d <+80>:    cmp    eax,edx
   0x000000000040072f <+82>:    je     0x400738 <check_password+91>
   0x0000000000400731 <+84>:    mov    eax,0x0
   0x0000000000400736 <+89>:    jmp    0x400749 <check_password+108>
   0x000000000040071b <+62>:    mov    edx,DWORD PTR [rbp-0x8]
<SNIP>
```

We can see that `rcx` is being added to the counter (`rbp-0x8`), so maybe `rcx` contains a pointer to the scrambled password, let's take a look:

```
(gdb) x/s $rcx
0x4008bf:       "ejezpvibwfgvo"
```

Looks promising! Let substract 1 from all characters and we get:

didyouhavefun

Run the program again and try it out:

```
$ ./level2
Password: didyouhavefun
Correct password!
```