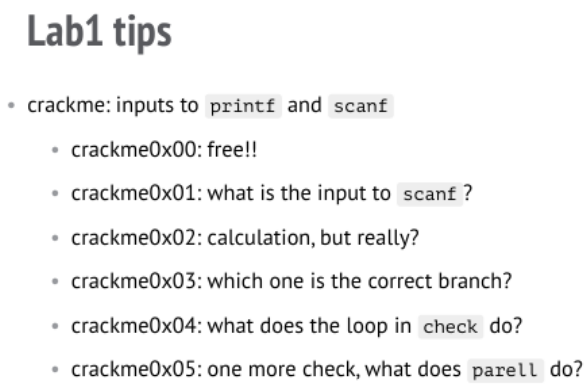
CS255 Lab 1: Reverse Engineering – Write Up

Hugo Wan, twan012

Due Date: 10/14/2025



- Here are the answers and explanations in short for graders:

1. Crackme0x00 tutorial is completed. Get the address that says what the password is directly. The password is **250382** here.
2. Crackme0x01's input to scanf is **5247**. It was found in the address where it accepts the password when the value is 0x149a in hex = 5274 in decimal.
3. Crackme0x02's calculation is **338724**, following the assembly code:

   local_c = 0x5a (90), local_10 = 0x1ec (492)

   → local_c += local_10 = 90 + 492 = 582

   → eax = local_c; eax *= local_c = 582 * 582 = 338724
4. Crackme0x03's correct branch is "**0x0804847a <test+12>: je 0x0804848a <test+28>"** because "je" means jump if the user input and the calculated password are equal. If the random user input is not **338724**, then it would show "Invalid Password!" after going through the test and the shift functions.
5. Crackme0x04's loop in the check function iteratively reads each character of the user's input, converts it into a number using sscanf ("%d", &num), and adds that value to an accumulator variable. After every addition, the program compares the running total to 0xF (15). **If the sum equals 15 at any point, the program immediately prints "Password OK!"** and exits successfully.
6. Crackme0x05's parell function converts the current digit from the password into an integer and **checks its parity using a bitmask (and $1)**. If the number is **even**, it prints "Password Incorrect!" and terminates the program.

- The full answers of how I got them are listed below:

1. Crackme0x00:
   a. Following the tutorial instructions:

      First, I loaded the crackme0x00 binary into GDB and ran it to the point where it prompts for a password. Then. input “r” to run the program.

      ```
      /Labs/Lab 1/IOLI-crackme$ gdb ./crackme0x00
      (gdb) r
      ```

   b. While the program waited for input, I interrupted execution with Ctrl+C (SIGINT). Paused the program inside the standard library call that reads input (scanf), allowing me to inspect the call stack and program state by entering the “bt” command.

      ```
      IOLI Crackme Level 0x00
      Password: ^C
      Program received signal SIGINT, Interrupt.
      0xf7fc7569 in __kernel_vsyscall ()
      (gdb) bt
      #0  0xf7fc7569 in __kernel_vsyscall ()
      #1  0xf7e96a93 in read () from /lib32/libc.so.6
      #2  0xf7e055c8 in _IO_file_underflow () from /lib32/libc.so.6
      #3  0xf7e07b46 in _IO_default_uflow () from /lib32/libc.so.6
      #4  0xf7de61c9 in ?? () from /lib32/libc.so.6
      #5  0xf7de0d65 in scanf () from /lib32/libc.so.6
      #6  0x0804845b in main ()
      ```

   c. Next, I set a temporary breakpoint at 0x0804845b, which is the instruction immediately after the scanf call. This location is ideal because the program has finished reading the user input and stored it in the local buffer, but the string comparison has not yet executed. The “tbreak” command stops execution when the address is reached and then removes itself. Lastly, after continuing “c” and entering sample input “aaa…”, GDB paused at the temporary breakpoint.

      ```
      (gdb) tbreak *0x0804845b
      Temporary breakpoint 1 at 0x804845b
      (gdb) c
      Continuing.
      aaaaaaaaaa

      Temporary breakpoint 1, 0x0804845b in main ()
      ```

d. I disassembled the main function with the “disas” command.

```
(gdb) disas
Dump of assembler code for function main:
   0x08048414 <+0>:     push   %ebp
   0x08048415 <+1>:     mov    %esp,%ebp
   0x08048417 <+3>:     sub    $0x28,%esp
   0x0804841a <+6>:     and    $0xfffffff0,%esp
   0x0804841d <+9>:     mov    $0x0,%eax
   0x08048422 <+14>:    add    $0xf,%eax
   0x08048425 <+17>:    add    $0xf,%eax
   0x08048428 <+20>:    shr    $0x4,%eax
   0x0804842b <+23>:    shl    $0x4,%eax
   0x0804842e <+26>:    sub    %eax,%esp
   0x08048430 <+28>:    movl   $0x8048568,(%esp)
   0x08048437 <+35>:    call   0x8048340 <printf@plt>
   0x0804843c <+40>:    movl   $0x8048581,(%esp)
   0x08048443 <+47>:    call   0x8048340 <printf@plt>
   0x08048448 <+52>:    lea    -0x18(%ebp),%eax
   0x0804844b <+55>:    mov    %eax,0x4(%esp)
   0x0804844f <+59>:    movl   $0x804858c,(%esp)
   0x08048456 <+66>:    call   0x8048330 <scanf@plt>
=> 0x0804845b <+71>:    lea    -0x18(%ebp),%eax
   0x0804845e <+74>:    movl   $0x804858f,0x4(%esp)
   0x08048466 <+82>:    mov    %eax,(%esp)
   0x08048469 <+85>:    call   0x8048350 <strcmp@plt>
   0x0804846e <+90>:    test   %eax,%eax
   0x08048470 <+92>:    je     0x8048480 <main+108>
   0x08048472 <+94>:    movl   $0x8048596,(%esp)
   0x08048479 <+101>:   call   0x8048340 <printf@plt>
   0x0804847e <+106>:   jmp    0x804848c <main+120>
   0x08048480 <+108>:   movl   $0x80485a9,(%esp)
   0x08048487 <+115>:   call   0x8048340 <printf@plt>
   0x0804848c <+120>:   mov    $0x0,%eax
   0x08048491 <+125>:   leave
   0x08048492 <+126>:   ret
End of assembler dump.
```

e. I inspected the relevant memory addresses with x /s to reveal the format string, the input buffer contents, and the hard-coded password value.

```
(gdb) x /s 0x804858c
0x804858c:      "%s"
(gdb) x /s $ebp-0x18
0xffffce00:     "aaaaaaaaaa"
(gdb) x /s 0x804858f
0x804858f:      "250382"
(gdb) x /s 0x8048596
0x8048596:      "Invalid Password!\n"
(gdb) x /s 0x80485a9
0x80485a9:      "Password OK :)\n"
```

f. From this evidence the program's logic is clear:

```
scanf("%s", local_buf);
if (!strcmp(local_buf, "250382"))
    printf("Password OK :)\n");
else
    printf("Invalid Password!\n");
```

g. Therefore, the correct password for crackme0x00 is **250382**, which I verified by running the binary outside the debugger. Otherwise, the password is invalid.

```
Lab 1/IOLI-crackme$ ./crackme0x00
IOLI Crackme Level 0x00
Password: 123456
Invalid Password!
```

```
Lab 1/IOLI-crackme$ ./crackme0x00
IOLI Crackme Level 0x00
Password: 250382
Password OK :)
```

2. Crackme0x01:

a. Quickly get to what we have:

```
/Labs/Lab 1/IOLI-crackme$ gdb ./crackme0x01
```

```
(gdb) r
IOLI Crackme Level 0x01
Password: ^C
Program received signal SIGINT, Interrupt.
0xf7fc7569 in __kernel_vsyscall ()
(gdb) bt
#0  0xf7fc7569 in __kernel_vsyscall ()
#1  0xf7e96a93 in read () from /lib32/libc.so.6
#2  0xf7e055c8 in _IO_file_underflow () from /lib32/libc.so.6
#3  0xf7e07b46 in _IO_default_uflow () from /lib32/libc.so.6
#4  0xf7de61c9 in ?? () from /lib32/libc.so.6
#5  0xf7de0d65 in scanf () from /lib32/libc.so.6
#6  0x0804842b in main ()
(gdb) tbreak *0x0804842b
Temporary breakpoint 1 at 0x804842b
(gdb) c
Continuing.
aaaaaaaaaaaaaaaaaaa

Temporary breakpoint 1, 0x0804842b in main ()
```

```
(gdb) disas
Dump of assembler code for function main:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $0x18,%esp
   0x080483ea <+6>:     and    $0xfffffff0,%esp
   0x080483ed <+9>:     mov    $0x0,%eax
   0x080483f2 <+14>:    add    $0xf,%eax
   0x080483f5 <+17>:    add    $0xf,%eax
   0x080483f8 <+20>:    shr    $0x4,%eax
   0x080483fb <+23>:    shl    $0x4,%eax
   0x080483fe <+26>:    sub    %eax,%esp
   0x08048400 <+28>:    movl   $0x8048528,(%esp)
   0x08048407 <+35>:    call   0x804831c <printf@plt>
   0x0804840c <+40>:    movl   $0x8048541,(%esp)
   0x08048413 <+47>:    call   0x804831c <printf@plt>
   0x08048418 <+52>:    lea    -0x4(%ebp),%eax
   0x0804841b <+55>:    mov    %eax,0x4(%esp)
   0x0804841f <+59>:    movl   $0x804854c,(%esp)
   0x08048426 <+66>:    call   0x804830c <scanf@plt>
=> 0x0804842b <+71>:    cmpl   $0x149a,-0x4(%ebp)
   0x08048432 <+78>:    je     0x8048442 <main+94>
   0x08048434 <+80>:    movl   $0x804854f,(%esp)
   0x0804843b <+87>:    call   0x804831c <printf@plt>
   0x08048440 <+92>:    jmp    0x804844e <main+106>
   0x08048442 <+94>:    movl   $0x8048562,(%esp)
   0x08048449 <+101>:   call   0x804831c <printf@plt>
   0x0804844e <+106>:   mov    $0x0,%eax
   0x08048453 <+111>:   leave
   0x08048454 <+112>:   ret
End of assembler dump.
```

b. I found something different than previous binary – scanf uses "%d" here at 0x0804854c, which means that the password should be an integer, not “%s” for a string. In experiment, I tried both x /s <addr> and x /d <addr>, and it shows “” and 0, respectively. This is because I typed “aaaa…” after the “c” continuing command. With "%d", that causes scanf to fail to parse and it doesn’t write to the local int – so it stays 0 or unchanged. Furthermore. I tried entering x /s and x /d at 0x149a but couldn’t access the memory for either one. Then, I realized this 0x149a may be the correct password for me to try outside the debugger. Lastly, I found the addresses that print out the result of the password.

```
(gdb) x /s 0x804854c
0x804854c:      "%d"
(gdb) x /s $ebp-0x4
0xffffce14:     ""
(gdb) x /d $ebp-0x4
0xffffce14:     0
(gdb) x /s 0x149a
0x149a: <error: Cannot access memory at address 0x149a>
(gdb) x /d 0x149a
0x149a: Cannot access memory at address 0x149a
(gdb) x /s 0x804854f
0x804854f:      "Invalid Password!\n"
(gdb) x /s 0x8048562
0x8048562:      "Password OK :)\n"
```

c. I was incorrect about the password, thinking that it was the hex 0x"149a".

```
/Labs/Lab 1/IOLI-crackme$ ./crackme0x01
IOLI Crackme Level 0x01
Password: 149a
Invalid Password!
```

d. The correct password should be in decimal. Here is the calculation:

x149A = 1 * 16^3 + 4 * 16^2 + 9 * 16 ^ 1 + 10 = **5274.**

```
/Labs/Lab 1/IOLI-crackme$ ./crackme0x01
IOLI Crackme Level 0x01
Password: 5274
Password OK :)
```

3. Crackme0x02:

a. Run the program in gdb::

```
/Labs/Lab 1/IOLI-crackme$ gdb ./crackme0x02
(gdb) r
```

```
IOLI Crackme Level 0x02
Password: ^C
Program received signal SIGINT, Interrupt.
0xf7fc7569 in __kernel_vsyscall ()
(gdb) bt
#0  0xf7fc7569 in __kernel_vsyscall ()
#1  0xf7e96a93 in read () from /lib32/libc.so.6
#2  0xf7e055c8 in _IO_file_underflow () from /lib32/libc.so.6
#3  0xf7e07b46 in _IO_default_uflow () from /lib32/libc.so.6
#4  0xf7de61c9 in ?? () from /lib32/libc.so.6
#5  0xf7de0d65 in scanf () from /lib32/libc.so.6
#6  0x0804842b in main ()
(gdb) tbreak *0x0804842b
Temporary breakpoint 1 at 0x804842b
(gdb) c
Continuing.
55555

Temporary breakpoint 1, 0x0804842b in main ()
```

```
(gdb) disas
Dump of assembler code for function main:
   0x080483e4 <+0>:     push   %ebp
   0x080483e5 <+1>:     mov    %esp,%ebp
   0x080483e7 <+3>:     sub    $0x18,%esp
   0x080483ea <+6>:     and    $0xfffffff0,%esp
   0x080483ed <+9>:     mov    $0x0,%eax
   0x080483f2 <+14>:    add    $0xf,%eax
   0x080483f5 <+17>:    add    $0xf,%eax
   0x080483f8 <+20>:    shr    $0x4,%eax
   0x080483fb <+23>:    shl    $0x4,%eax
   0x080483fe <+26>:    sub    %eax,%esp
   0x08048400 <+28>:    movl   $0x8048548,(%esp)
   0x08048407 <+35>:    call   0x804831c <printf@plt>
   0x0804840c <+40>:    movl   $0x8048561,(%esp)
   0x08048413 <+47>:    call   0x804831c <printf@plt>
   0x08048418 <+52>:    lea    -0x4(%ebp),%eax
   0x0804841b <+55>:    mov    %eax,0x4(%esp)
   0x0804841f <+59>:    movl   $0x804856c,(%esp)
   0x08048426 <+66>:    call   0x804830c <scanf@plt>
=> 0x0804842b <+71>:    movl   $0x5a,-0x8(%ebp)
   0x08048432 <+78>:    movl   $0x1ec,-0xc(%ebp)
   0x08048439 <+85>:    mov    -0xc(%ebp),%edx
   0x0804843c <+88>:    lea    -0x8(%ebp),%eax
   0x0804843f <+91>:    add    %edx,(%eax)
   0x08048441 <+93>:    mov    -0x8(%ebp),%eax
   0x08048444 <+96>:    imul   -0x8(%ebp),%eax
   0x08048448 <+100>:   mov    %eax,-0xc(%ebp)
   0x0804844b <+103>:   mov    -0x4(%ebp),%eax
   0x0804844e <+106>:   cmp    -0xc(%ebp),%eax
   0x08048451 <+109>:   jne    0x8048461 <main+125>
   0x08048453 <+111>:   movl   $0x804856f,(%esp)
   0x0804845a <+118>:   call   0x804831c <printf@plt>
   0x0804845f <+123>:   jmp    0x804846d <main+137>
   0x08048461 <+125>:   movl   $0x804857f,(%esp)
   0x08048468 <+132>:   call   0x804831c <printf@plt>
   0x0804846d <+137>:   mov    $0x0,%eax
   0x08048472 <+142>:   leave
   0x08048473 <+143>:   ret
End of assembler dump.
```

b. Now, look for the type of password. It's "%d" – so it's an integer again. Also, it shows what I typed before.

```
(gdb) x /s 0x804856c
0x804856c:      "%d"
(gdb) x /dw $ebp-0x4
0xffffce14:     55555
```

c. After some deep exploration of commands and deep understanding of the assembly, I found the password after calculation:

local_c = 0x5a (90), local_10 = 0x1ec (492)

→ local_c += local_10 = 90 + 492 = 582

→ eax = local_c; eax *= local_c = 582 * 582 = 338724

```
(gdb) p/d 0x5a
$10 = 90
(gdb) p/d 0x1ec
$11 = 492
(gdb) p/d 0x5a + 0x1ec
$12 = 582
(gdb) p/d 582 * 582
$13 = 338724
```

d. Then, I ran the program with **338724**. It is correct as the password.

```
IOLI Crackme Level 0x02
Password: 338724
Password OK :)
```

4. Crackme0x03:
   a. Main difference is that the random user input matters after setting up temporary breakpoint. If the user input is not **338724**, then it will show "Invalid Passord!" after testing through the test and shift functions

```
IOLI Crackme Level 0x03
Password: ^C
Program received signal SIGINT, Interrupt.
0xf7fc7569 in __kernel_vsyscall ()
(gdb) bt
#0  0xf7fc7569 in __kernel_vsyscall ()
#1  0xf7e96a93 in read () from /lib32/libc.so.6
#2  0xf7e055c8 in _IO_file_underflow () from /lib32/libc.so.6
#3  0xf7e07b46 in _IO_default_uflow () from /lib32/libc.so.6
#4  0xf7de61c9 in ?? () from /lib32/libc.so.6
#5  0xf7de0d65 in scanf () from /lib32/libc.so.6
#6  0x080484df in main ()
(gdb) tbreak *0x080484df
Temporary breakpoint 1 at 0x80484df
(gdb) c
Continuing.
338724

Temporary breakpoint 1, 0x080484df in main ()
```

b. Disassemble the main function:

```
(gdb) disas main
Dump of assembler code for function main:
   0x08048498 <+0>:     push   %ebp
   0x08048499 <+1>:     mov    %esp,%ebp
   0x0804849b <+3>:     sub    $0x18,%esp
   0x0804849e <+6>:     and    $0xfffffff0,%esp
   0x080484a1 <+9>:     mov    $0x0,%eax
   0x080484a6 <+14>:    add    $0xf,%eax
   0x080484a9 <+17>:    add    $0xf,%eax
   0x080484ac <+20>:    shr    $0x4,%eax
   0x080484af <+23>:    shl    $0x4,%eax
   0x080484b2 <+26>:    sub    %eax,%esp
   0x080484b4 <+28>:    movl   $0x8048610,(%esp)
   0x080484bb <+35>:    call   0x8048350 <printf@plt>
   0x080484c0 <+40>:    movl   $0x8048629,(%esp)
   0x080484c7 <+47>:    call   0x8048350 <printf@plt>
   0x080484cc <+52>:    lea    -0x4(%ebp),%eax
   0x080484cf <+55>:    mov    %eax,0x4(%esp)
   0x080484d3 <+59>:    movl   $0x8048634,(%esp)
   0x080484da <+66>:    call   0x8048330 <scanf@plt>
=> 0x080484df <+71>:    movl   $0x5a,-0x8(%ebp)
   0x080484e6 <+78>:    movl   $0x1ec,-0xc(%ebp)
   0x080484ed <+85>:    mov    -0xc(%ebp),%edx
   0x080484f0 <+88>:    lea    -0x8(%ebp),%eax
   0x080484f3 <+91>:    add    %edx,(%eax)
   0x080484f5 <+93>:    mov    -0x8(%ebp),%eax
   0x080484f8 <+96>:    imul   -0x8(%ebp),%eax
   0x080484fc <+100>:   mov    %eax,-0xc(%ebp)
   0x080484ff <+103>:   mov    -0xc(%ebp),%eax
   0x08048502 <+106>:   mov    %eax,0x4(%esp)
   0x08048506 <+110>:   mov    -0x4(%ebp),%eax
   0x08048509 <+113>:   mov    %eax,(%esp)
   0x0804850c <+116>:   call   0x804846e <test>
   0x08048511 <+121>:   mov    $0x0,%eax
   0x08048516 <+126>:   leave
   0x08048517 <+127>:   ret
End of assembler dump.
```

c. Password type: “%d”, an integer. The user input: 338724.

```
(gdb) x/s 0x8048634
0x8048634:      "%d"
(gdb) x/dw $ebp-0x4
0xffffce14:     338724
```

d. Now, look for the password to compare with the user input by setting another temporary breakpoint after the calculation in main function. The password should be **338724**.

```
(gdb) tbreak *0x080484ff
Temporary breakpoint 2 at 0x80484ff
(gdb) c
Continuing.

Temporary breakpoint 2, 0x080484ff in main ()
(gdb) x/i $pc
=> 0x80484ff <main+103>:        mov    -0xc(%ebp),%eax
(gdb) x/dw $ebp-0xc
0xffffce0c:     338724
```

e. These are the values before calling test function for comparison.

a. $ebp-0x4 is the user input from the 1st breakpoint.

b. $ebp-0xc is the calculated password from the 2nd breakpoint.

```
(gdb) x/dw $ebp-0x4
0xffffce14:     338724
(gdb) x/dw $ebp-0xc
0xffffce0c:     338724
```

f. Setting another temporary breakpoint at the start of the test function. Enter "finish" run through both test and shift functions to get the results of the compared inputs:

```
(gdb) tbreak *0x804846e
Temporary breakpoint 3 at 0x804846e
(gdb) c
Continuing.

Temporary breakpoint 3, 0x0804846e in test ()
(gdb) x/i $pc
=> 0x804846e <test>:    push   %ebp
(gdb) finish
Run till exit from #0  0x0804846e in test ()
Password OK!!! :)
0x08048511 in main ()
```

g. Taking a closer look at the test function. The correct branch that leads to "Password OK 😊" is **"0x0804847a <test+12>: je 0x0804848a <test+28>"** because "je" means jump if the user input and the calculated password are equal.

```
(gdb) disas test
Dump of assembler code for function test:
   0x0804846e <+0>:     push   %ebp
   0x0804846f <+1>:     mov    %esp,%ebp
   0x08048471 <+3>:     sub    $0x8,%esp
   0x08048474 <+6>:     mov    0x8(%ebp),%eax
   0x08048477 <+9>:     cmp    0xc(%ebp),%eax
   0x0804847a <+12>:    je     0x804848a <test+28>
   0x0804847c <+14>:    movl   $0x80485ec,(%esp)
   0x08048483 <+21>:    call   0x8048414 <shift>
   0x08048488 <+26>:    jmp    0x8048496 <test+40>
   0x0804848a <+28>:    movl   $0x80485fe,(%esp)
   0x08048491 <+35>:    call   0x8048414 <shift>
   0x08048496 <+40>:    leave
   0x08048497 <+41>:    ret
End of assembler dump.
```

h. Here are the results tested outside of the GNU Debugger:

```
Lab 1/IOLI-crackme$ ./crackme0x03
IOLI Crackme Level 0x03
Password: 55555
Invalid Password!
```

```
Lab 1/IOLI-crackme$ ./crackme0x03
IOLI Crackme Level 0x03
Password: 338724
Password OK!!! :)
```

5. Crackme0x04:

a. Starting with the temporary breakpoint to main function again.

```
IOLI Crackme Level 0x04
Password: ^C
Program received signal SIGINT, Interrupt.
0xf7fc7569 in __kernel_vsyscall ()
(gdb) bt
#0  0xf7fc7569 in __kernel_vsyscall ()
#1  0xf7e96a93 in read () from /lib32/libc.so.6
#2  0xf7e055c8 in _IO_file_underflow () from /lib32/libc.so.6
#3  0xf7e07b46 in _IO_default_uflow () from /lib32/libc.so.6
#4  0xf7de61c9 in ?? () from /lib32/libc.so.6
#5  0xf7de0d65 in scanf () from /lib32/libc.so.6
#6  0x08048553 in main ()
(gdb) tbreak *0x08048553
Temporary breakpoint 1 at 0x8048553
(gdb) c
Continuing.
96

Temporary breakpoint 1, 0x08048553 in main ()
```

b. Disassemble the main function:

```
(gdb) disas
Dump of assembler code for function main:
   0x08048509 <+0>:     push   %ebp
   0x0804850a <+1>:     mov    %esp,%ebp
   0x0804850c <+3>:     sub    $0x88,%esp
   0x08048512 <+9>:     and    $0xfffffff0,%esp
   0x08048515 <+12>:    mov    $0x0,%eax
   0x0804851a <+17>:    add    $0xf,%eax
   0x0804851d <+20>:    add    $0xf,%eax
   0x08048520 <+23>:    shr    $0x4,%eax
   0x08048523 <+26>:    shl    $0x4,%eax
   0x08048526 <+29>:    sub    %eax,%esp
   0x08048528 <+31>:    movl   $0x804865e,(%esp)
   0x0804852f <+38>:    call   0x8048394 <printf@plt>
   0x08048534 <+43>:    movl   $0x8048677,(%esp)
   0x0804853b <+50>:    call   0x8048394 <printf@plt>
   0x08048540 <+55>:    lea    -0x78(%ebp),%eax
   0x08048543 <+58>:    mov    %eax,0x4(%esp)
   0x08048547 <+62>:    movl   $0x8048682,(%esp)
   0x0804854e <+69>:    call   0x8048374 <scanf@plt>
=> 0x08048553 <+74>:    lea    -0x78(%ebp),%eax
   0x08048556 <+77>:    mov    %eax,(%esp)
   0x08048559 <+80>:    call   0x8048484 <check>
   0x0804855e <+85>:    mov    $0x0,%eax
   0x08048563 <+90>:    leave
   0x08048564 <+91>:    ret
End of assembler dump.
```

c. Password type: “%s”, a string. The random user input is “96” in char.

```
(gdb) x /s 0x8048682
0x8048682:      "%s"
(gdb) x /s $ebp-0x78
0xffffcda0:     "96"
```

d. Set another temporary breakpoint at the start of the check function:

```
(gdb) tbreak *0x8048484
Temporary breakpoint 2 at 0x8048484
(gdb) c
Continuing.

Temporary breakpoint 2, 0x08048484 in check ()
```

```
(gdb) disas check
Dump of assembler code for function check:
=> 0x08048484 <+0>:     push   %ebp
   0x08048485 <+1>:     mov    %esp,%ebp
   0x08048487 <+3>:     sub    $0x28,%esp
   0x0804848a <+6>:     movl   $0x0,-0x8(%ebp)
   0x08048491 <+13>:    movl   $0x0,-0xc(%ebp)
   0x08048498 <+20>:    mov    0x8(%ebp),%eax
   0x0804849b <+23>:    mov    %eax,(%esp)
   0x0804849e <+26>:    call   0x8048384 <strlen@plt>
   0x080484a3 <+31>:    cmp    %eax,-0xc(%ebp)
   0x080484a6 <+34>:    jae    0x80484fb <check+119>
   0x080484a8 <+36>:    mov    -0xc(%ebp),%eax
   0x080484ab <+39>:    add    0x8(%ebp),%eax
   0x080484ae <+42>:    movzbl (%eax),%eax
   0x080484b1 <+45>:    mov    %al,-0xd(%ebp)
   0x080484b4 <+48>:    lea    -0x4(%ebp),%eax
   0x080484b7 <+51>:    mov    %eax,0x8(%esp)
   0x080484bb <+55>:    movl   $0x8048638,0x4(%esp)
   0x080484c3 <+63>:    lea    -0xd(%ebp),%eax
   0x080484c6 <+66>:    mov    %eax,(%esp)
   0x080484c9 <+69>:    call   0x80483a4 <sscanf@plt>
   0x080484ce <+74>:    mov    -0x4(%ebp),%edx
   0x080484d1 <+77>:    lea    -0x8(%ebp),%eax
   0x080484d4 <+80>:    add    %edx,(%eax)
   0x080484d6 <+82>:    cmpl   $0xf,-0x8(%ebp)
   0x080484da <+86>:    jne    0x80484f4 <check+112>
   0x080484dc <+88>:    movl   $0x804863b,(%esp)
   0x080484e3 <+95>:    call   0x8048394 <printf@plt>
   0x080484e8 <+100>:   movl   $0x0,(%esp)
   0x080484ef <+107>:   call   0x80483b4 <exit@plt>
   0x080484f4 <+112>:   lea    -0xc(%ebp),%eax
   0x080484f7 <+115>:   incl   (%eax)
   0x080484f9 <+117>:   jmp    0x8048498 <check+20>
   0x080484fb <+119>:   movl   $0x8048649,(%esp)
   0x08048502 <+126>:   call   0x8048394 <printf@plt>
   0x08048507 <+131>:   leave
   0x08048508 <+132>:   ret
```

e. In the check function, the loop inside the check function iteratively reads each character of the user's input, converts it into a number using sscanf ("%d", &num), and adds that value to an accumulator variable. After every addition, the program compares the running total to 0xF (15). If the sum equals 15 at any point, the program immediately prints "Password OK!" and exits successfully. Otherwise, it continues looping through the remaining characters. Non-digit characters contribute 0 to the total.

f. Here are some passwords tested outside the GNU Debugger.

Any password that once accumulates to digit 15 is the correct password.

a. “338724” = 3+3+8” = 14+”7” = 21 > 15, results in “Password Incorrect!”

b. “1058010” = 1+0+5+8+0+1+0 = 15, results in “Password OK!”

c. “5a5b5c5d5e”, non-digits are 0 in numbers. 5+0+5+0+5 = 15, which also results in “Password OK!”

```
Lab 1/IOLI-crackme$ ./crackme0x04
IOLI Crackme Level 0x04
Password: 338724
Password Incorrect!
```

```
Lab 1/IOLI-crackme$ ./crackme0x04
IOLI Crackme Level 0x04
Password: 1058010
Password OK!
```

```
Lab 1/IOLI-crackme$ ./crackme0x04
IOLI Crackme Level 0x04
Password: 5a5b5c5d5e
Password OK!
```

6. Crackme0x05:

a. Starting with the temporary breakpoint to main function:

```
IOLI Crackme Level 0x05
Password: ^C
Program received signal SIGINT, Interrupt.
0xf7fc7569 in __kernel_vsyscall ()
(gdb) bt
#0  0xf7fc7569 in __kernel_vsyscall ()
#1  0xf7e96a93 in read () from /lib32/libc.so.6
#2  0xf7e055c8 in _IO_file_underflow () from /lib32/libc.so.6
#3  0xf7e07b46 in _IO_default_uflow () from /lib32/libc.so.6
#4  0xf7de61c9 in ?? () from /lib32/libc.so.6
#5  0xf7de0d65 in scanf () from /lib32/libc.so.6
#6  0x0804858a in main ()
(gdb) tbreak *0x0804858a
Temporary breakpoint 1 at 0x804858a
(gdb) c
Continuing.
196

Temporary breakpoint 1, 0x0804858a in main ()
```

b. Disassemble the main function:

```
(gdb) disas
Dump of assembler code for function main:
   0x08048540 <+0>:     push   %ebp
   0x08048541 <+1>:     mov    %esp,%ebp
   0x08048543 <+3>:     sub    $0x88,%esp
   0x08048549 <+9>:     and    $0xfffffff0,%esp
   0x0804854c <+12>:    mov    $0x0,%eax
   0x08048551 <+17>:    add    $0xf,%eax
   0x08048554 <+20>:    add    $0xf,%eax
   0x08048557 <+23>:    shr    $0x4,%eax
   0x0804855a <+26>:    shl    $0x4,%eax
   0x0804855d <+29>:    sub    %eax,%esp
   0x0804855f <+31>:    movl   $0x804868e,(%esp)
   0x08048566 <+38>:    call   0x8048394 <printf@plt>
   0x0804856b <+43>:    movl   $0x80486a7,(%esp)
   0x08048572 <+50>:    call   0x8048394 <printf@plt>
   0x08048577 <+55>:    lea    -0x78(%ebp),%eax
   0x0804857a <+58>:    mov    %eax,0x4(%esp)
   0x0804857e <+62>:    movl   $0x80486b2,(%esp)
   0x08048585 <+69>:    call   0x8048374 <scanf@plt>
=> 0x0804858a <+74>:    lea    -0x78(%ebp),%eax
   0x0804858d <+77>:    mov    %eax,(%esp)
   0x08048590 <+80>:    call   0x80484c8 <check>
   0x08048595 <+85>:    mov    $0x0,%eax
   0x0804859a <+90>:    leave
   0x0804859b <+91>:    ret
End of assembler dump.
```

c. Password type: “%s”, a string. The user input: “196” in char.

```
(gdb) x /s 0x80486b2
0x80486b2:      "%s"
(gdb) x /s $ebp-0x78
0xffffcda0:     "196"
```

d. Disassemble the check function:

```
(gdb) disas check
Dump of assembler code for function check:
   0x080484c8 <+0>:     push   %ebp
   0x080484c9 <+1>:     mov    %esp,%ebp
   0x080484cb <+3>:     sub    $0x28,%esp
   0x080484ce <+6>:     movl   $0x0,-0x8(%ebp)
   0x080484d5 <+13>:    movl   $0x0,-0xc(%ebp)
   0x080484dc <+20>:    mov    0x8(%ebp),%eax
   0x080484df <+23>:    mov    %eax,(%esp)
   0x080484e2 <+26>:    call   0x8048384 <strlen@plt>
   0x080484e7 <+31>:    cmp    %eax,-0xc(%ebp)
   0x080484ea <+34>:    jae    0x8048532 <check+106>
   0x080484ec <+36>:    mov    -0xc(%ebp),%eax
   0x080484ef <+39>:    add    0x8(%ebp),%eax
   0x080484f2 <+42>:    movzbl (%eax),%eax
   0x080484f5 <+45>:    mov    %al,-0xd(%ebp)
   0x080484f8 <+48>:    lea    -0x4(%ebp),%eax
   0x080484fb <+51>:    mov    %eax,0x8(%esp)
   0x080484ff <+55>:    movl   $0x8048668,0x4(%esp)
   0x08048507 <+63>:    lea    -0xd(%ebp),%eax
   0x0804850a <+66>:    mov    %eax,(%esp)
   0x0804850d <+69>:    call   0x80483a4 <sscanf@plt>
   0x08048512 <+74>:    mov    -0x4(%ebp),%edx
   0x08048515 <+77>:    lea    -0x8(%ebp),%eax
   0x08048518 <+80>:    add    %edx,(%eax)
   0x0804851a <+82>:    cmpl   $0x10,-0x8(%ebp)
   0x0804851e <+86>:    jne    0x804852b <check+99>
   0x08048520 <+88>:    mov    0x8(%ebp),%eax
   0x08048523 <+91>:    mov    %eax,(%esp)
   0x08048526 <+94>:    call   0x8048484 <parell>
   0x0804852b <+99>:    lea    -0xc(%ebp),%eax
   0x0804852e <+102>:   incl   (%eax)
   0x08048530 <+104>:   jmp    0x80484dc <check+20>
   0x08048532 <+106>:   movl   $0x8048679,(%esp)
   0x08048539 <+113>:   call   0x8048394 <printf@plt>
   0x0804853e <+118>:   leave
   0x0804853f <+119>:   ret
End of assembler dump.
```

e. The check function loops through each digit, adding it to a running total. The program only prints “Password OK!” when this total reaches **16**, and it survives all calls to parell before that point. Therefore, the final valid passwords are those whose digits **sum to 16** and **do not trigger the early exit** in parell.

f. Disassemble the parell function:

```
(gdb) disas parell
Dump of assembler code for function parell:
   0x08048484 <+0>:     push   %ebp
   0x08048485 <+1>:     mov    %esp,%ebp
   0x08048487 <+3>:     sub    $0x18,%esp
   0x0804848a <+6>:     lea    -0x4(%ebp),%eax
   0x0804848d <+9>:     mov    %eax,0x8(%esp)
   0x08048491 <+13>:    movl   $0x8048668,0x4(%esp)
   0x08048499 <+21>:    mov    0x8(%ebp),%eax
   0x0804849c <+24>:    mov    %eax,(%esp)
   0x0804849f <+27>:    call   0x80483a4 <sscanf@plt>
   0x080484a4 <+32>:    mov    -0x4(%ebp),%eax
   0x080484a7 <+35>:    and    $0x1,%eax
   0x080484aa <+38>:    test   %eax,%eax
   0x080484ac <+40>:    jne    0x80484c6 <parell+66>
   0x080484ae <+42>:    movl   $0x804866b,(%esp)
   0x080484b5 <+49>:    call   0x8048394 <printf@plt>
   0x080484ba <+54>:    movl   $0x0,(%esp)
   0x080484c1 <+61>:    call   0x80483b4 <exit@plt>
   0x080484c6 <+66>:    leave
   0x080484c7 <+67>:    ret
End of assembler dump.
```

g. The parell function converts the current digit from the password into an integer and checks its parity using a bitmask (and $1). If the number is even, it prints “Password Incorrect!” and terminates the program.

h. Here are some working and incorrect password tested outside GDB:

```
/Labs/Lab 1/IOLI-crackme$ ./crackme0x05
IOLI Crackme Level 0x05
Password: 79
Password Incorrect!
```

```
/Labs/Lab 1/IOLI-crackme$ ./crackme0x05
IOLI Crackme Level 0x05
Password: 196
Password OK!
```