CS 255 Lab1 Reverse Engineering

Name: Vedant Deepak Borkute
SID: 862552981
NetID: vbork001

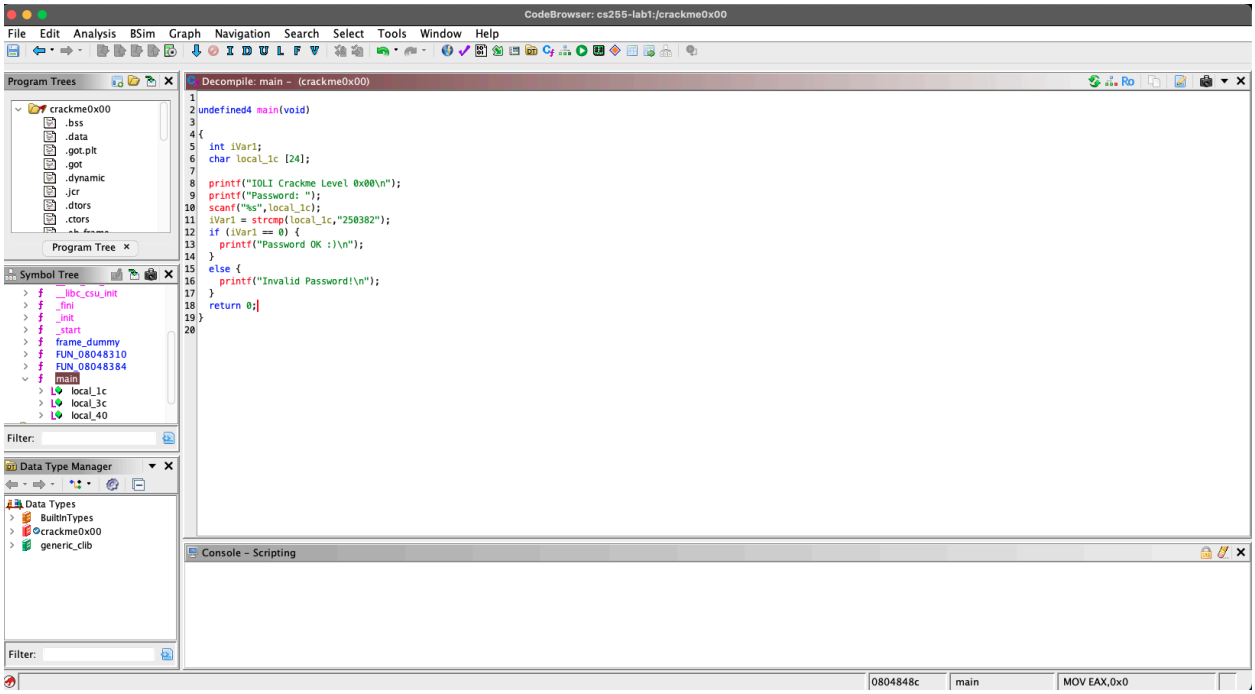
1. crackme0x00
   Running objdump -d on crackme0x00, going thru the resulting assembly code, in the main(),we are getting a string input (by checking the type of argument passed to it at `0x804858c` and later comparing that input with a value at address `0x0804858f` to determine whether the password was correct. When I ran gdb with the binary, I used `x/s 0x0804858f` to get the string in that address, which was **<u>250382</u>**.

```
(gdb) x/s 0x0804858f
0x804858f:      "250382"
(gdb)
```

```
objdump_1.txt
cs255-f2025 > L1 > objdump_output > objdump_1.txt

crackme0x00:     file format elf32-i386

Disassembly of section .init:

080482f8 <_init>:
 80482f8: 55                            pushl   %ebp
 80482f9: 89 e5                         movl    %esp, %ebp
 80482fb: 83 ec 08                      subl    $0x8, %esp
 80482fe: e8 81 00 00 00                calll   0x8048384 <_start+0x24>
 8048303: e8 d8 00 00 00                calll   0x80483e0 <frame_dummy>
 8048308: e8 13 02 00 00                calll   0x8048520 <__do_global_ctors_aux>
 804830d: c9                            leave
 804830e: c3                            retl

Disassembly of section .plt:

08048310 <.plt>:
 8048310: ff 35 f8 9f 04 08             pushl   0x8049ff8
 8048316: ff 25 fc 9f 04 08             jmpl    *0x8049ffc
 804831c: 00 00                         addb    %al, (%eax)
 804831e: 00 00                         addb    %al, (%eax)

08048320 <__libc_start_main@plt>:
 8048320: ff 25 00 a0 04 08             jmpl    *0x804a000
 8048326: 68 00 00 00 00                pushl   $0x0
 804832b: e9 e0 ff ff ff                jmp 0x8048310 <.plt>

08048330 <scanf@plt>:
 8048330: ff 25 04 a0 04 08             jmpl    *0x804a004
 8048336: 68 08 00 00 00                pushl   $0x8
 804833b: e9 d0 ff ff ff                jmp 0x8048310 <.plt>

08048340 <printf@plt>:
 8048340: ff 25 08 a0 04 08             jmpl    *0x804a008
 8048346: 68 10 00 00 00                pushl   $0x10
 804834b: e9 c0 ff ff ff                jmp 0x8048310 <.plt>

08048350 <strcmp@plt>:
 8048350: ff 25 0c a0 04 08             jmpl    *0x804a00c
 8048356: 68 18 00 00 00                pushl   $0x18
 804835b: e9 b0 ff ff ff                jmp 0x8048310 <.plt>

Disassembly of section .text:
```

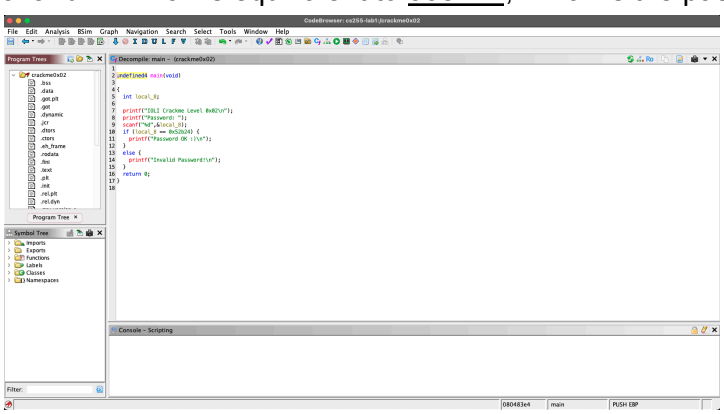
```
EndUser@MacBookPro binaries % docker run -v ./:/binaries -w /binaries -e LOLO+= -it --rm i386/ubuntu:bionic ./crackme0x00
IOLI Crackme Level 0x00
Password: 250382
Password OK :)
```

2. crackme0x01
   Using the decompiled code from Ghidra, we see that the input (integer) is being compared to the hex number 0x149a which is **<u>5274</u>**, if it matches, it prints “Password OK”.




```
EndUser@MacBookPro binaries % docker run -v ./:/binaries -w /binaries -e LOLO+= -it --rm i386/ubuntu:bionic ./crackme0x01
IOLI Crackme Level 0x01
Password: 5274
Password OK :)
```

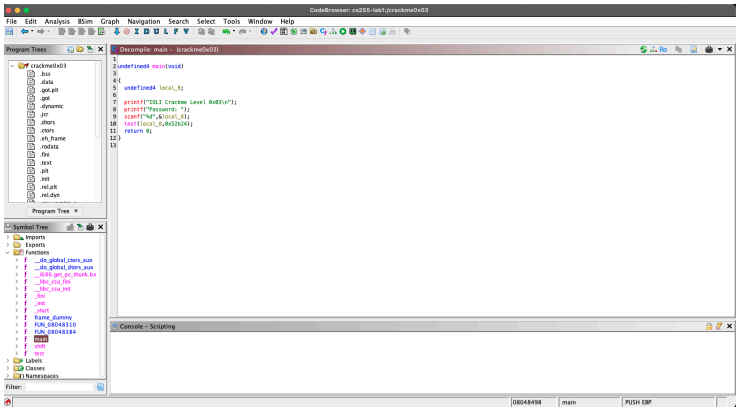
3. crackme0x02
   In the decompiled code, the integer input variable is compared with the hex number 0x52b24 which is equivalent to **338724**, which is the password.



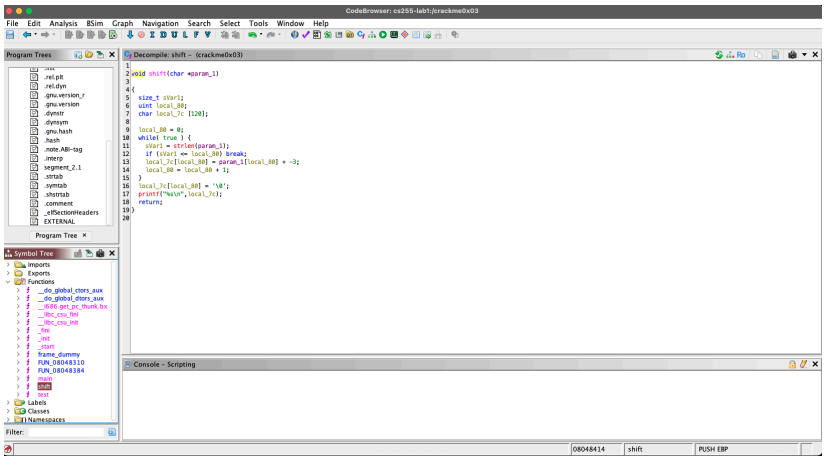
```
EndUser@MacBookPro binaries % docker run -v ./:/binaries -w /binaries -it --rm i386/ubuntu:bionic ./crackme0x02
IOLI Crackme Level 0x02
Password: 338724
Password OK :)
```

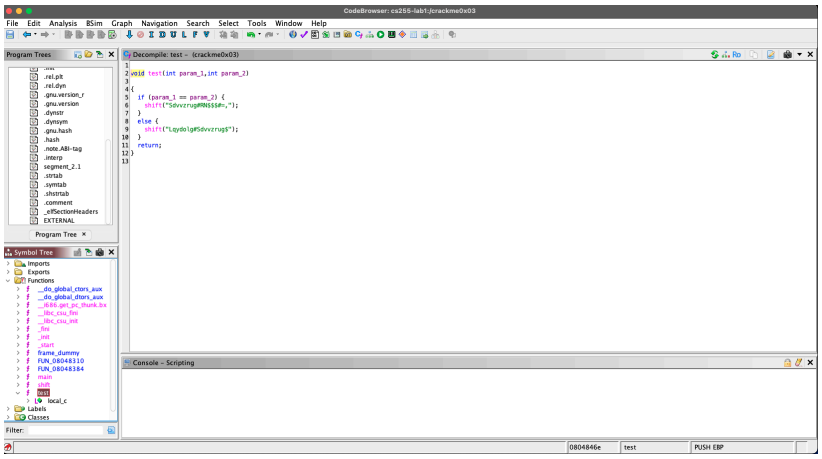
4. crackme0x03
   In the decompiled code, an integer input is taken, then test() is called with two parameters, the input and the hex number 0x52b24 (338724). The test(), if the two params are same, calls shift() for the string Sdvvzrug#RN$$$#=, which then forms another string from this string, shifted by -3. If they aren't, the shift function is called for

“Lqydolg#Sdvvzrug$”. Decoding these strings, I find that the first string refers to “Passwork OK case”. Therefore the password for this binary is still **<u>338724</u>**.



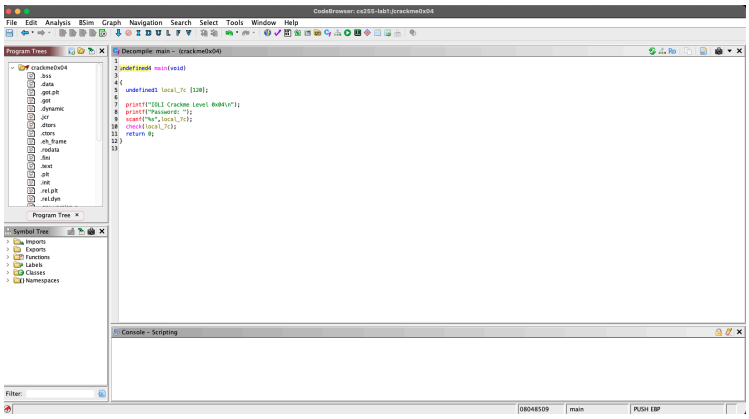

```
EndUser@MacBookPro binaries % docker run -v ./:/binaries -w /binaries -it --rm i386/ubuntu:bionic ./crackme0x03
IOLI Crackme Level 0x03
Password: 338724
Password OK!!! :)
```

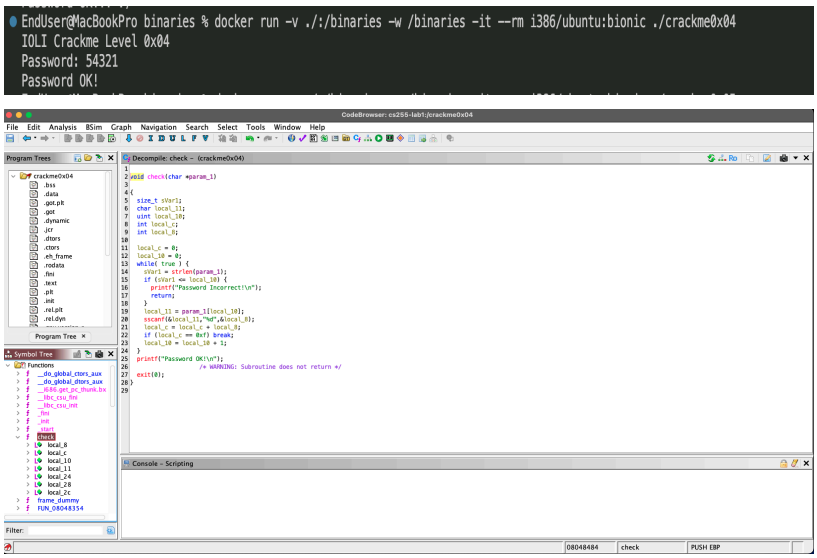




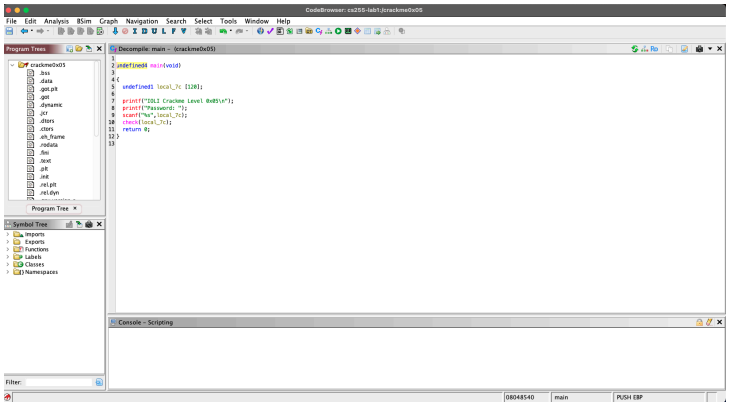
5. crackme0x04
   Using the decompiler, this time the code is looking for a string input and passing it to the check(), which then iteratively attempts to compute the running total of integers in the string. If the total reaches 0xf (15), the loop is broken. If the total is not yet reached and

the whole string has been traversed, it prints out password incorrect. Also, if the total is greater than 15, it prints out password incorrect. But if the string contains non-numeric characters, along with integers that sum to 15, it won't work. An example password that works is "**54321**".

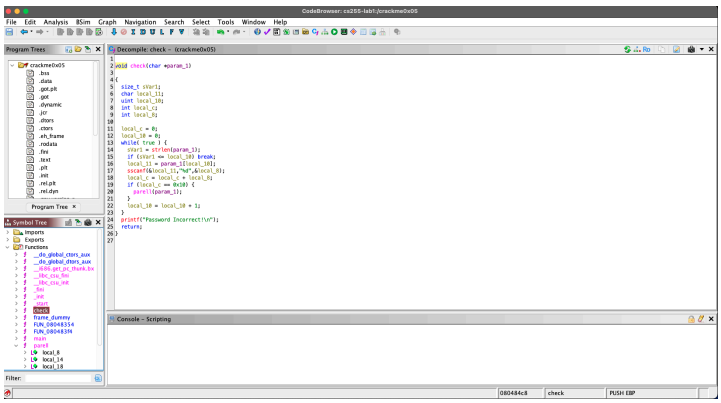
6. 

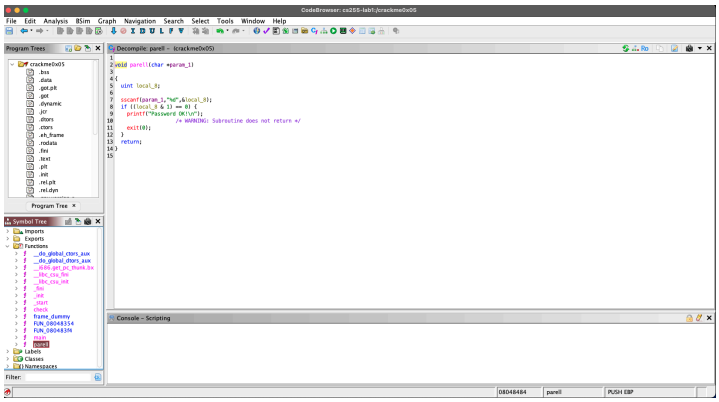


7. crackme0x05
   Again, using the decompiler, its taking an input string and passing it to check(), which is doing a similar thing as to last time, except that the total is supposed to reach 0x10 (16). If the total is that, parell() is called on the input, where a bitwise and operation to check if the number is even or odd is done. Only an even number summing to 16 is accepted. Therefore a valid password is **5434**.



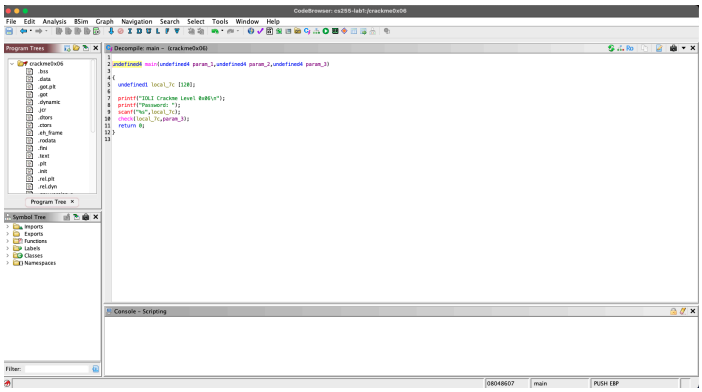
```
EndUser@MacBookPro binaries % docker run -v ./:/binaries -w /binaries -it --rm i386/ubuntu:bionic ./crackme0x05
IOLI Crackme Level 0x05
Password: 5434
Password OK!
```



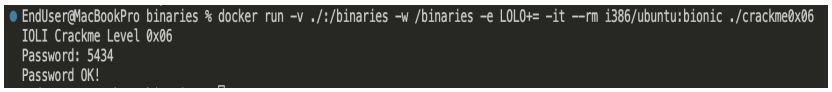


8. crackme0x06
   Similar to previous binaries, a string input is taken. Then check() is checking the running total of the integers in the string. If the total has reached 16, then parell() is called with the input as param_1 and param_2 (which are env vars). Here, the dummy function is called with the integer of the input and the param_2. It now loops through the list of strings in param_2 to find any starting with “LOL”. If there are, dummy returns 1. Then, parell() finally checks if the integer formed from the input string is even, if yes, it prints Password OK.

To provide the program the env variable containing “LOL”, we can run the binary as ./crackme0x06 LOLO=+ and then provide a password which sums to 16 and is even like



“**<u>5434</u>**”.



CodeBrowser: cs255-lab1:/crackme0x06

Decompile: check - (crackme0x06)

```
void check(char *param_1,undefined4 param_2)

{
  size_t sVar1;
  char local_11;
  uint local_10;
  int local_c;
  int local_8;

  local_c = 0;
  local_10 = 0;
  while( true ) {
    sVar1 = strlen(param_1);
    if (sVar1 <= local_10) break;
    local_11 = param_1[local_10];
    sscanf(&local_11,"%d",&local_8);
    local_c = local_c + local_8;
    if (local_c == 0x10) {
      parell(param_1,param_2);
    }
    local_10 = local_10 + 1;
  }
  printf("Password Incorrect!\n");
  return;
}
```

Console - Scripting

08048588 | check | PUSH EBP

CodeBrowser: cs255-lab1:/crackme0x06

Decompile: parell - (crackme0x06)

```
void parell(char *param_1,undefined4 param_2)

{
  int iVar1;
  int local_c;
  uint local_8;

  sscanf(param_1,"%d",&local_8);
  iVar1 = dummy(local_8,param_2);
  if (iVar1 != 0) {
    for (local_c = 0; local_c < 10; local_c = local_c + 1) {
      if ((local_8 & 1) == 0) {
        printf("Password OK!\n");
                    /* WARNING: Subroutine does not return */
        exit(0);
      }
    }
  }
  return;
}
```

Console - Scripting

0804851a | parell | PUSH EBP

CodeBrowser: cs255-lab1:/crackme0x06

Decompile: dummy - (crackme0x06)

```
undefined4 dummy(undefined4 param_1,int param_2)

{
  int iVar1;
  int local_8;

  local_8 = 0;
  do {
    if (*(int *)(local_8 * 4 + param_2) == 0) {
      return 0;
    }
    iVar1 = local_8 * 4;
    local_8 = local_8 + 1;
    iVar1 = strncmp(*(char **)(iVar1 + param_2),"LOLO",3);
  } while (iVar1 != 0);
  return 1;
}
```

Console - Scripting

080484b4 | dummy | PUSH EBP