Assignment 4 - BASC

BINARY ANALYSIS AND SECURE CODING - UniGE

Kevin Cattaneo - S4944382

Index

Extract

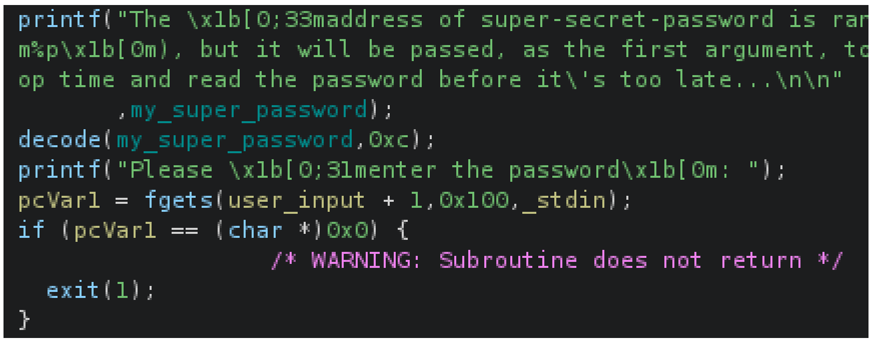
“This assignment aims to familiarize yourself with advanced static-analysis techniques and tools.
In AulaWeb you'll find the-locks_student_binaries.tar.xz, containing two different executables for each student (that has linked their account with our GitHub classroom).
As file names suggest, the difficulty of "picking a lock" (=finding the correct password to get a flag back) grows as the level increases.”

The Lock - Level 1

Tool used: pwntools, Ghidra

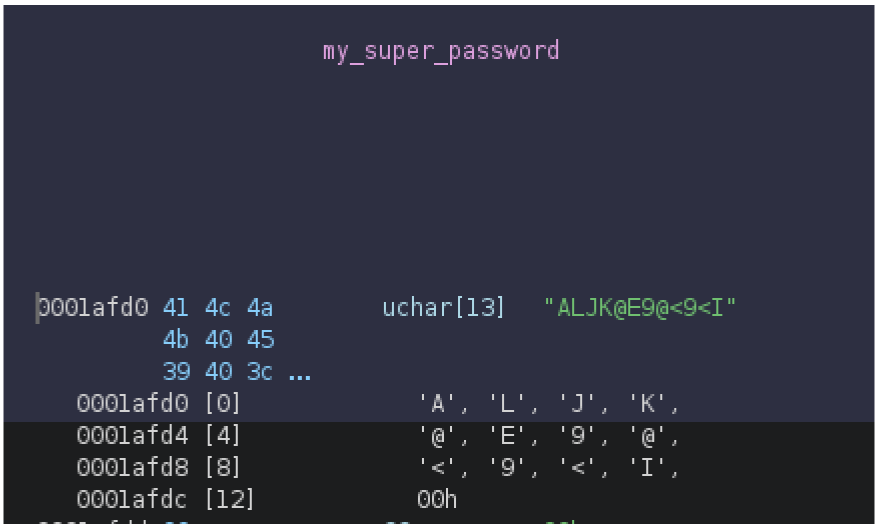
The lock is a challenge that asks you for a secret password to unlock the flag. That password is stored into a random address that is leaked on the console, together with other leaks. This password is internally decoded and then compared with the user input, then its address content is cleared with zeros.
What I need to do is to statically analyse the program, so without running the program apart the run to retrieve the flag upon finding the password.

By looking with Ghidra, I discover the position of the decode function, in the *main*:



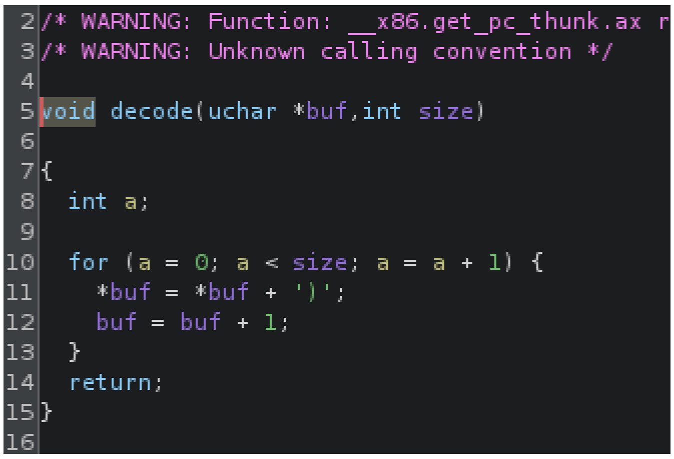
```
printf("The \x1b[0;33maddress of super-secret-password is ran
m%p\x1b[0m), but it will be passed, as the first argument, to
op time and read the password before it\'s too late...\n\n"
       ,my_super_password);
decode(my_super_password,0xc);
printf("Please \x1b[0;31menter the password\x1b[0m: ");
pcVar1 = fgets(user_input + 1,0x100,_stdin);
if (pcVar1 == (char *)0x0) {
                  /* WARNING: Subroutine does not return */
  exit(1);
}
```

The my_super_password address holds the password encrypted:



```
                    my_super_password

0001afd0 41 4c 4a        uchar[13]   "ALJK@E9@<9<I"
         4b 40 45
         39 40 3c ...
   0001afd0 [0]            'A', 'L', 'J', 'K',
   0001afd4 [4]            '@', 'E', '9', '@',
   0001afd8 [8]            '<', '9', '<', 'I',
   0001afdc [12]           00h
```

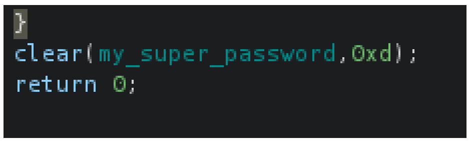
While the super difficult decoding function is the following:



```
/* WARNING: Function: __x86.get_pc_thunk.ax r
/* WARNING: Unknown calling convention */

void decode(uchar *buf,int size)

{
  int a;

  for (a = 0; a < size; a = a + 1) {
    *buf = *buf + ')';
    buf = buf + 1;
  }
  return;
}
```

That just adds the byte value of the char ‘)’.

While the clearing of the password address content is performed at the very end (but since I don’t have to execute the program is not so scary):



```
}
clear(my_super_password,0xd);
return 0;
```

So with pwntools I just try to read the memory address and retrieve the encoded password. In alternative, since the executable is not stripped, I can directly read the content of the symbol *my_super_password*.
Then, since it is very simple, I reproduce the decoding function into my script and then print the decoded password:

```
Password: justinbieber
```

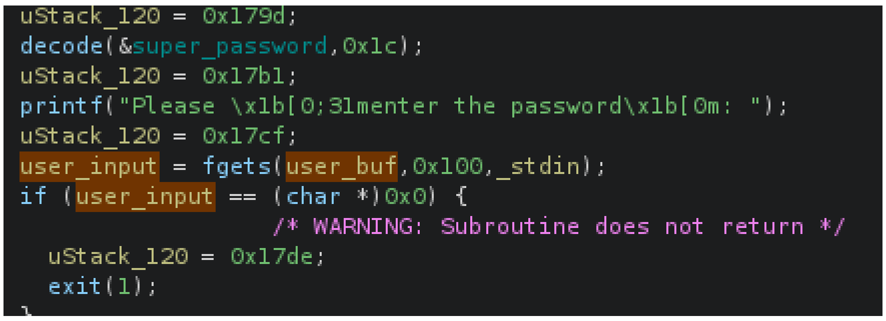
After finding the password I run the executable just to print the flag, passing the decoded password:

```
Please enter the password:
Wow! You got it, congratulations.
Here is your flag:
BASC{Y0u_int3rc3pt3d_stRcMp_didnt_U---keatane-YMdQYT3c}
```

The Lock - Level 2

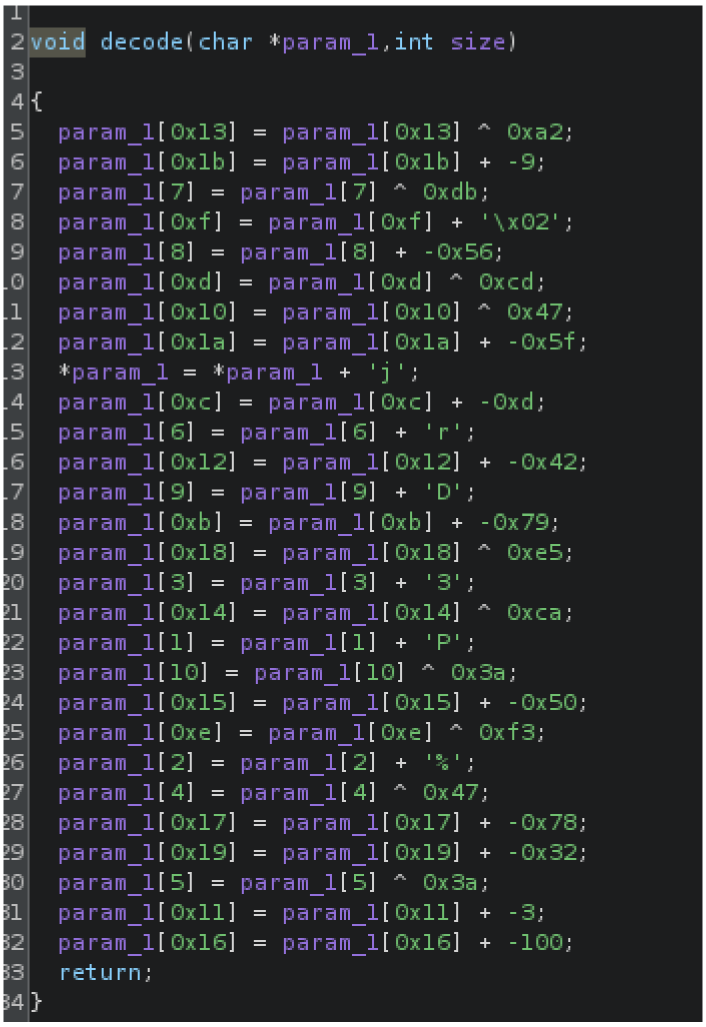
Tool used: pwntools, unicorn, Ghidra

The level 2 script is visually the same as level 1, but this time the binary is stripped, so no symbols and also is a 64-bit executable.
I again look at Ghidra, renaming stuff that I easily recognize from the level 1 counterpart:



```
uStack_120 = 0x179d;
decode(&super_password,0x1c);
uStack_120 = 0x17b1;
printf("Please \x1b[0;31menter the password\x1b[0m: ");
uStack_120 = 0x17cf;
user_input = fgets(user_buf,0x100,_stdin);
if (user_input == (char *)0x0) {
                  /* WARNING: Subroutine does not return */
  uStack_120 = 0x17de;
  exit(1);
}
```

But this time the decode function is on another level (literally):



```

void decode(char *param_1,int size)

{
  param_1[0x13] = param_1[0x13] ^ 0xa2;
  param_1[0x1b] = param_1[0x1b] + -9;
  param_1[7] = param_1[7] ^ 0xdb;
  param_1[0xf] = param_1[0xf] + '\x02';
  param_1[8] = param_1[8] + -0x56;
  param_1[0xd] = param_1[0xd] ^ 0xcd;
  param_1[0x10] = param_1[0x10] ^ 0x47;
  param_1[0x1a] = param_1[0x1a] + -0x5f;
  *param_1 = *param_1 + 'j';
  param_1[0xc] = param_1[0xc] + -0xd;
  param_1[6] = param_1[6] + 'r';
  param_1[0x12] = param_1[0x12] + -0x42;
  param_1[9] = param_1[9] + 'D';
  param_1[0xb] = param_1[0xb] + -0x79;
  param_1[0x18] = param_1[0x18] ^ 0xe5;
  param_1[3] = param_1[3] + '3';
  param_1[0x14] = param_1[0x14] ^ 0xca;
  param_1[1] = param_1[1] + 'P';
  param_1[10] = param_1[10] ^ 0x3a;
  param_1[0x15] = param_1[0x15] + -0x50;
  param_1[0xe] = param_1[0xe] ^ 0xf3;
  param_1[2] = param_1[2] + '%';
  param_1[4] = param_1[4] ^ 0x47;
  param_1[0x17] = param_1[0x17] + -0x78;
  param_1[0x19] = param_1[0x19] + -0x32;
  param_1[5] = param_1[5] ^ 0x3a;
  param_1[0x11] = param_1[0x11] + -3;
  param_1[0x16] = param_1[0x16] + -100;
  return;
}
```

I could use ChatGPT to translate that into Python, but since I want to learn, I used unicorn to emulate the decoding 🙂

So I start creating a new script with pwntools where I perform the initialization of the code space and the stack with the mapping, just like in the lesson.

```
PG_CODE_SIZE = (CODE_SIZE + 4095) & ~4095
emu.mem_map(CODE_ADDR, PG_CODE_SIZE, UC_PROT_ALL)
emu.mem_write(CODE_ADDR, all_bytes)
emu.mem_map(STACK_ADDR, STACK_SIZE, UC_PROT_READ|UC_PROT_WRITE)
emu.reg_write(UC_X86_REG_RSP, STACK_ADDR + STACK_SIZE)
```

Then I read the encoded content of the address of *my_super_password* (`0x00017fd0`) and write it into the RDI register, to pass it to the decode function as an argument.

```
emu.reg_write(UC_X86_REG_RDI, 0x00017fd0)
```

After that I start the emulation of the decode function:
```
emu.emu_start(0x000011f3, 0x00001498)
```

Then I read again the address saved into the RDI register that has been changed in place by the decode function, decode it as a string:

```
emu_dec_data = emu.mem_read(0x00017fd0,0x1c).decode('latin1')
```

Decoded Password: projectsadminx-international

Once obtained the password, I start again the executable just to pass it and obtain the flag:
Wow! You got it, congratulations.
Here is your flag:
BASC{Br3akP0int5_and_3mul4t10n_R_us3fUl---keatane-tZBeuY05}

Extra: Level 0 - Confusion

The very first start of that assignment was with the wrong foot. Indeed I focused myself into parsing the output of the challenge, retrieving the address where the secret password was stored, reading its content via GDB and then decode it. That was caused by the fact that I need to stop in some way that executable before clearing the address content, so before anything I tried to retrieve that value.
Indeed I completed the challenge because the password *justinbieber* was found.

Still I launched GDB so it was not fair, because the assignment required *static analysis* of the code.

So after an afternoon spent on something wrong, at least I was already warmed up for the real challenge.

I left the wrong.py script with the tentative in GDB just as a trace of work, that still is not fully working in an automatic way, but I had already spent too much time on something non useful.
(The regex was generated)

```
p = process('./keatane-the_lock-level_1')
p.recvuntil(b'(this time it is \x1b[0;35m')
address = p.recvuntil(b')')
address = address.strip()
address = address.split(b'\x1b')[0]
password_address = int(address, 16)
print(hex(password_address))

gdb_output = ''
gdb.attach(p, gdbscript='''
     print (char*){hex(password_address)}
     quit
''')

gdb_output = p.recvline()

print("GDB Output:")
print(gdb_output)
clean_string = re.sub(r'\x1b\[[0-9;]*m', '',
gdb_output.decode('utf-8')).strip()
print(clean_string)
```