Lab 3
Ghidra Software Reverse Engineering

Prepared for: Dr. Collin O'Flynn

Abdulla Sadoun B00900541

Sohaib Zahid B00908166

Table of Contents

Table of Contents **1**
Introduction and Objective **2**
Procedure **2**
Questions **2**
Question 1: **2**
Question 2: **5**
Question 3: **7**
Question 4: 7
Question 5: **9**
Conclusion **12**

Introduction and Objective

This lab aims to explore reverse engineering techniques using Ghidra, a software analysis tool. By analyzing a provided hex file (would usually be binary in real applications without the starting address but we have it for this lab), we will utilize Ghidra's features, such as disassembly, memory mapping, and peripheral interaction as well as decompiling.

The key objectives are to configure Ghidra, load and work with System View Description (SVD) files, and perform automatic analysis of the hex file. This analysis will provide insights into how the program configures GPIO, handles peripherals, and manages password verification logic.

Procedure

We first installed the Ghidra application with the javascript feature already pre loaded intio it in our zip file.

We then uploaded the .hex file which would usually have the program data as well as instructions in binary format (machine language). Usually we would need to identify the starting address and pass it in as a parameter but this step wasn't needed for this lab as we already have the starting address.

We then proceeded to disassemble and decompile the code, finding the assembly code and decompiled c code, we also used the defined strings function in the code to give us a hint about what's being inputted and outputted in and from the code, also showing us the purpose of certain functions.

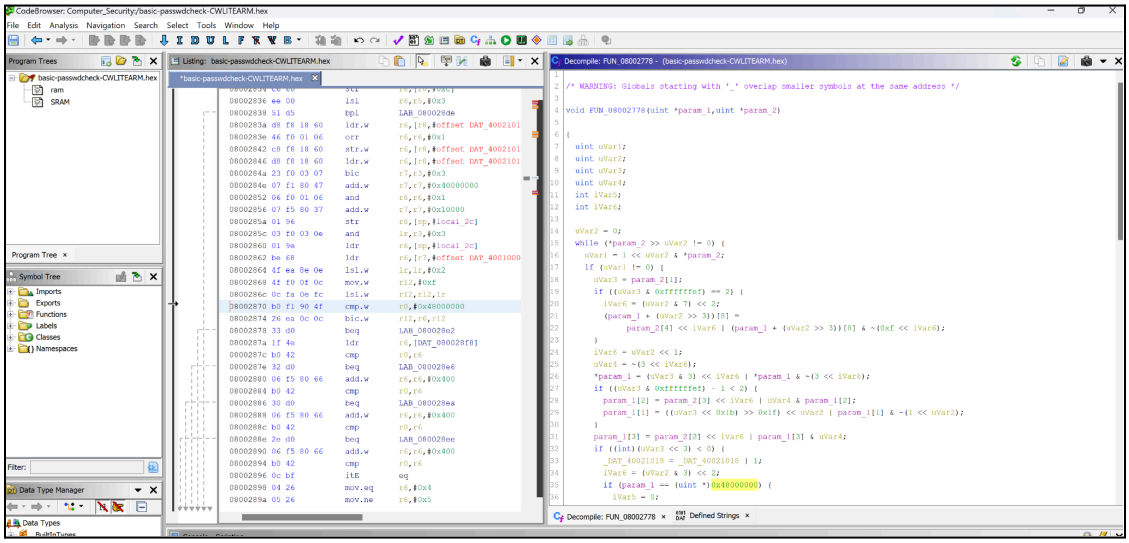
We also used the SVD loader to import the STM32 SVD file to define the memory-mapped registers for the microcontroller peripherals which aided in the process by labeling where peripherals would be in memory etc.

Questions

Question 1:

What function (i.e., what address) appears to configure GPIOA for use? In your lab report, include a snippet of the function and annotate what it is doing. Note there are multiple configuration locations, so you may have different answers for this. [10 pts]

It seems like the following function is the one responsible for configuring GPIOA for use:



```
3
4  void FUN_08002778(uint *param_1,uint *param_2)
5
6  {
7    uint uVar1;
8    uint uVar2;
9    uint uVar3;
10   uint uVar4;
11   int iVar5;
12   int iVar6;
13
14   uVar2 = 0;
15   while (*param_2 >> uVar2 != 0) {
16     uVar1 = 1 << uVar2 & *param_2;
17     if (uVar1 != 0) {
18       uVar3 = param_2[1];
```

The function 08002778 (function address in memory) seems to be the one configuring the GPIOA for use, this is said because after referring to the STM32 documentation and looking through the SRAM, We saw that the key register address check at 0x48000000 which corresponds to GPIOA in the documents is mentioned and used in the function:

```
void FUN_08002778(uint *param_1,uint *param_2)
```

At the following line in the function:

```
if (param_1 == (uint *)0x48000000) {
        iVar5 = 0;
    }
```

The function seems to take primarily two arguments, one for the GPIO structure and the other is for an array that contains the configuration settings for the corresponding GPIO.

In the function, If the following condition is met (`(uVar3 & 0xffffffef) == 2`), the function configures the code shifts and masks bits to configure which alternate function the pin should use on the GPIOA.

The following lines seem to be responsible for configuring the GPIO pin's mode (Input/Output/Alternate Function mode/Analog Mode) determined by the second element in the second parameter (array containing the settings).

uVar4 = ~(3 << iVar6): // line to set configuration for GPIO

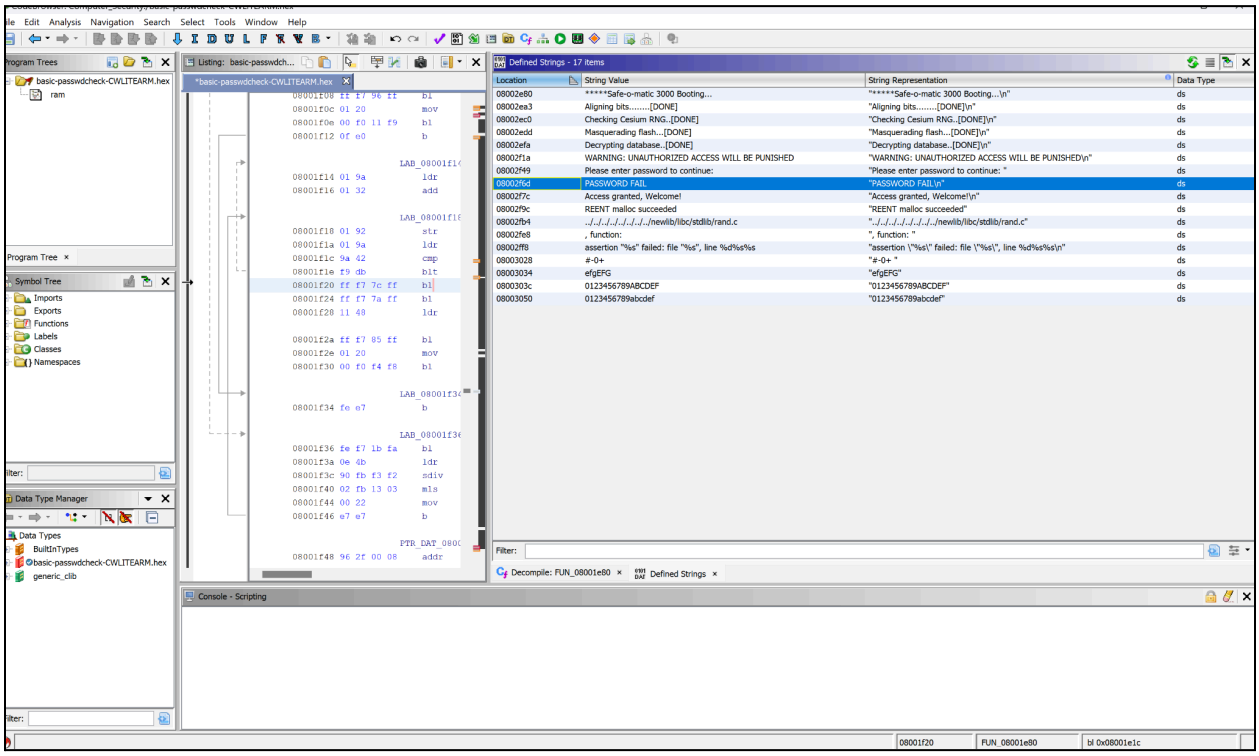
If output is chosen, an output speed and type can be specified. If the clock configuration is equipped further settings can be set by referring to the register in the parameters used.
Annotated function:

```
void FUN_08002778(uint *param_1, uint *param_2) {

   uint uVar1;
   uint uVar2 = 0;

   while (*param_2 >> uVar2 != 0) {
      uVar1 = 1 << uVar2 & *param_2;
      if (uVar1 != 0) {

         // Alternate Function Configuration
         uint uVar3 = param_2[1];
         if ((uVar3 & 0xffffffef) == 2) {
            int iVar6 = (uVar2 & 7) << 2;
            (param_1 + (uVar2 >> 3))[8] = param_2[4] << iVar6 | (param_1 + (uVar2 >> 3))[8] & ~(0xf << iVar6);
         }

         // GPIO Mode Configuration
         iVar6 = uVar2 << 1;
         uint uVar4 = ~(3 << iVar6);
         *param_1 = (uVar3 & 3) << iVar6 | *param_1 & ~(3 << iVar6);

         // Output Speed and Type Configuration
         if ((uVar3 & 0xffffffef) - 1 < 2) {
            param_1[2] = param_2[3] << iVar6 | uVar4 & param_1[2];
            param_1[1] = ((uVar3 << 0x1b) >> 0x1f) << uVar2 | param_1[1] & ~(1 << uVar2);
         }

         // Peripheral Clock Configuration
         if ((int)(uVar3 << 3) < 0) {
            _DAT_40021018 = _DAT_40021018 | 1;
         }
      }
      uVar2++;
   }
}
```

Question 2:

List all the defined strings in this program

The list of all the defined strings in the program can be found here:



Zooming in here are all the strings found with their addresses:

Location	String Value	String Representation	Data Type
08002e80	*****Safe-o-matic 3000 Booting...	"*****Safe-o-matic 3000 Booting...\n"	ds
08002ea3	Aligning bits.....[DONE]	"Aligning bits.....[DONE]\n"	ds
08002ec0	Checking Cesium RNG..[DONE]	"Checking Cesium RNG..[DONE]\n"	ds
08002edd	Masquerading flash...[DONE]	"Masquerading flash...[DONE]\n"	ds
08002efa	Decrypting database..[DONE]	"Decrypting database..[DONE]\n"	ds
08002f1a	WARNING: UNAUTHORIZED ACCESS WILL BE PUNISHED	"WARNING: UNAUTHORIZED ACCESS WILL BE PUNISHED\n"	ds
08002f49	Please enter password to continue:	"Please enter password to continue: "	ds
08002f6d	PASSWORD FAIL	"PASSWORD FAIL\n"	ds
08002f7c	Access granted, Welcome!	"Access granted, Welcome!\n"	ds
08002f9c	REENT malloc succeeded	"REENT malloc succeeded"	ds
08002fb4	../../../../../../../newlib/libc/stdlib/rand.c	"../../../../../../../newlib/libc/stdlib/rand.c"	ds
08002fe8	, function:	", function: "	ds
08002ff8	assertion "%s" failed: file "%s", line %d%s%s	"assertion \"%s\" failed: file \"%s\", line %d%s%s\n"	ds
08003028	#-0+	"#-0+ "	ds
08003034	efgEFG	"efgEFG"	ds
0800303c	0123456789ABCDEF	"0123456789ABCDEF"	ds
08003050	0123456789abcdef	"0123456789abcdef"	ds

String Value	String Representation
*****Safe-o-matic 3000 Booting...	"*****Safe-o-matic 3000 Booting...\n"
Aligning bits.....[DONE]	"Aligning bits.....[DONE]\n"
Checking Cesium RNG..[DONE]	"Checking Cesium RNG..[DONE]\n"
Masquerading flash...[DONE]	"Masquerading flash...[DONE]\n"
Decrypting database..[DONE]	"Decrypting database..[DONE]\n"
WARNING: UNAUTHORIZED ACCESS WILL BE PUNISHED	"WARNING: UNAUTHORIZED ACCESS WILL BE PUNISHED\n"
Please enter password to continue:	"Please enter password to continue: "
PASSWORD FAIL	"PASSWORD FAIL\n"
Access granted, Welcome!	"Access granted, Welcome!\n"
REENT malloc succeeded	"REENT malloc succeeded"
../../../../../../../../newlib/libc/stdlib/rand.c	"../../../../../../../../newlib/libc/stdlib/rand.c"
, function:	", function: "
assertion "%s" failed: file "%s", line %d%s%s	"assertion \"%s\" failed: file \"%s\", line %d%s%s\n"
#-0+	"#-0+ "
efgEFG	"efgEFG"
0123456789ABCDEF	"0123456789ABCDEF"
0123456789abcdef	"0123456789abcdef"

Question 3:

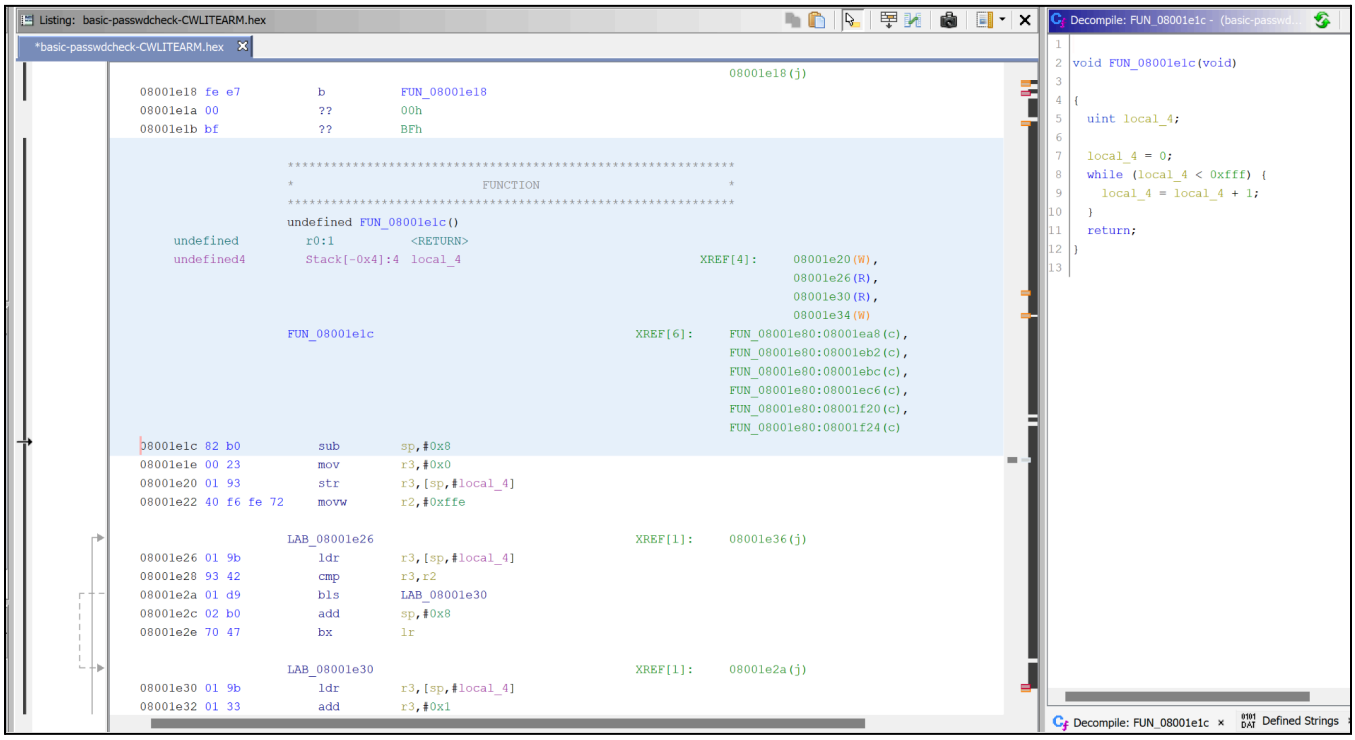
Does any peripheral in the code use the ADC1 or ADC2 peripheral? If so what is the address [2pts]

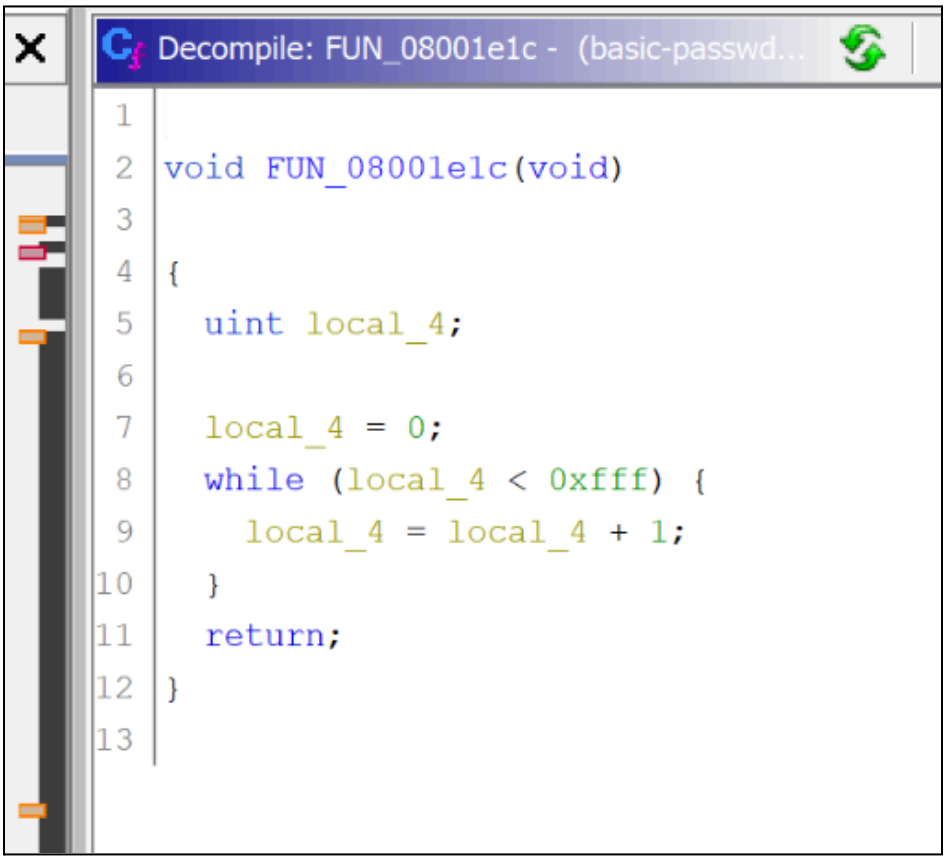
0x5000 0000 - 0x5000 03FF 1 K ADC1 - ADC2 - addresses for ADC1 and ADC 2 Peripherals from the STM memory map document, so I used cross-reference search to see if the program has used these base memory addresses which refer to ADC 1 and 2. So I looked in the current listing and saw that the current memories don't include this range so I added it from program memory with 0x50000000 as the base address and a length of 0x03FF to see if there is anything referencing these sections or any labels available which there wasn't.

This means that these peripherals probably were not used at all in this code.

Question 4:

What does the function at address 08001e1c do? Include a short annotation of the decompiled C code.





The function seems to be a simple delay loop function which are often used in embedded systems to wait for I/O devices to respond . It starts by creating (declaring) a variable and setting its value to 0, it then enters a loop which runs until the variable created is more than or equal to 0xFFF (0d4095) incrementing the variable by one every loop iteration until the condition becomes false then it just returns. Function Purpose:

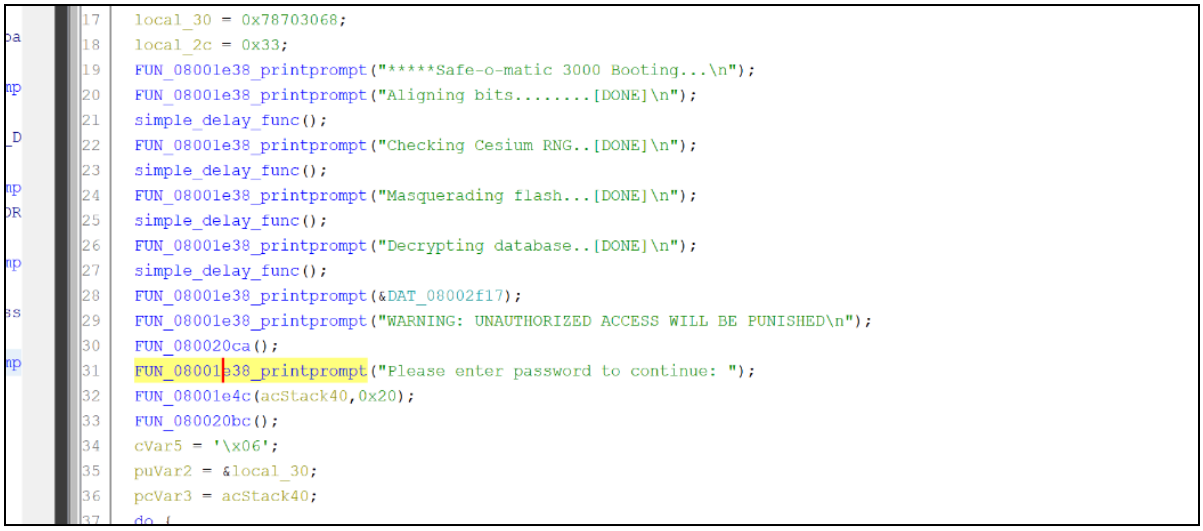
```
void FUN_08001e1c(void){ /annotated function
  uint local_4;
  local_4 = 0;  // Initialize local variable to 0
  while (local_4 < 0xfff) {  // Loop until local_4 reaches 0xFFF (4095 in decimal)
    local_4 = local_4 + 1;  // Increment local_4 in each iteration
  }
  return;  // End the function
}
```

Question 5:

For the password comparison logic: [10 pts]

a. Where is the password that is entered compared with the stored password? Include a short annotation of the password comparison logic. [8 pts]

To find out where the passwords are compared to check if the entry is right I first started exploring from the strings list and selected the “Please enter password to continue: \n” string to find where it is referenced which led me here.



```
17  local_30 = 0x78703068;
18  local_2c = 0x33;
19  FUN_08001e38_printprompt("*****Safe-o-matic 3000 Booting...\n");
20  FUN_08001e38_printprompt("Aligning bits.......[DONE]\n");
21  simple_delay_func();
22  FUN_08001e38_printprompt("Checking Cesium RNG..[DONE]\n");
23  simple_delay_func();
24  FUN_08001e38_printprompt("Masquerading flash...[DONE]\n");
25  simple_delay_func();
26  FUN_08001e38_printprompt("Decrypting database..[DONE]\n");
27  simple_delay_func();
28  FUN_08001e38_printprompt(&DAT_08002f17);
29  FUN_08001e38_printprompt("WARNING: UNAUTHORIZED ACCESS WILL BE PUNISHED\n");
30  FUN_080020ca();
31  FUN_08001e38_printprompt("Please enter password to continue: ");
32  FUN_08001e4c(acStack40,0x20);
33  FUN_080020bc();
34  cVar5 = '\x06';
35  puVar2 = &local_30;
36  pcVar3 = acStack40;
37  do {
```

So I labeled (renamed) that function for the prompt, which is located (called) in void FUN_08001e80(void).

After that function is called we get what looks like a function storing input and then some comparison under.

```
27  simple_delay_func();
28  FUN_08001e38_printprompt(&DAT_08002f17);
29  FUN_08001e38_printprompt("WARNING: UNAUTHORIZED ACCESS WILL BE PUNISHED\n");
30  FUN_080020ca();
31  FUN_08001e38_printprompt("Please enter password to continue: ");
32  FUN_08001e4c_usr_input(acStack40,0x20);
33  FUN_080020bc();
34  cVar5 = '\x06';
35  puVar2 = &local_30;
36  pcVar3 = acStack40;
37  do {
38    if (*(char *)puVar2 != *pcVar3) {
39      uVar1 = FUN_08000370();
40      iVar4 = 0;
41      while( true ) {
42        if ((int)(uVar1 % 100000) <= iVar4) break;
43        iVar4 = iVar4 + 1;
44      }
45      simple_delay_func();
46      simple_delay_func();
47      FUN_08001e38_printprompt("PASSWORD FAIL\n");
```

Then, we see the comparison at line 38 where local_30 (probably the password) is compared with acStack_28 (the user input) as it was mentioned in the previous uart user input function. This means that local_30 contains the password.
Annotated:

```
void FUN_08001e80(void)
{
  uint uVar1;
  undefined4 *puVar2;
  char *pcVar3;
  int iVar4;
  char cVar5;
  undefined4 local_30; // password declared
  undefined2 local_2c; // used to add a “3”
  char acStack40 [32];

  FUN_08001f78();
  FUN_08001ff8();
  FUN_08002074();
  local_30 = 0x78703068;
  local_2c = 0x33;
  FUN_08001e38_printprompt("*****Safe-o-matic 3000 Booting...\n");
  FUN_08001e38_printprompt("Aligning bits.....[DONE]\n");
  simple_delay_func();
  FUN_08001e38_printprompt("Checking Cesium RNG..[DONE]\n");
  simple_delay_func();
```

```
FUN_08001e38_printprompt("Masquerading flash...[DONE]\n");
simple_delay_func();
FUN_08001e38_printprompt("Decrypting database..[DONE]\n");
simple_delay_func();
FUN_08001e38_printprompt(&DAT_08002f17);
FUN_08001e38_printprompt("WARNING: UNAUTHORIZED ACCESS WILL BE PUNISHED\n");
FUN_080020ca();
FUN_08001e38_printprompt("Please enter password to continue: "); // prompt for password
FUN_08001e4c_usr_input(acStack40,0x20); // password inputted by user through USART
FUN_080020bc();
cVar5 = '\x06';
puVar2 = &local_30; // password set to this variable to check
pcVar3 = acStack40;
do {
  if (*(char *)puVar2 != *pcVar3) { // check if the password is correct
    uVar1 = FUN_08000370();
    iVar4 = 0;
    while( true ) {
      if ((int)(uVar1 % 100000) <= iVar4) break;
      iVar4 = iVar4 + 1;
    }
    simple_delay_func();
    simple_delay_func();
    FUN_08001e38_printprompt("PASSWORD FAIL\n"); // tell the user abt wrong attempt
    FUN_0800211c(1);
    goto LAB_08001f34;
  }
  cVar5 = cVar5 + -1;
  puVar2 = (undefined4 *)((int)puVar2 + 1);
  pcVar3 = pcVar3 + 1;
} while (cVar5 != '\0');
FUN_08001e38_printprompt("Access granted, Welcome!\n");
FUN_08002134(1);
```

b. What is the password? [1 pts]

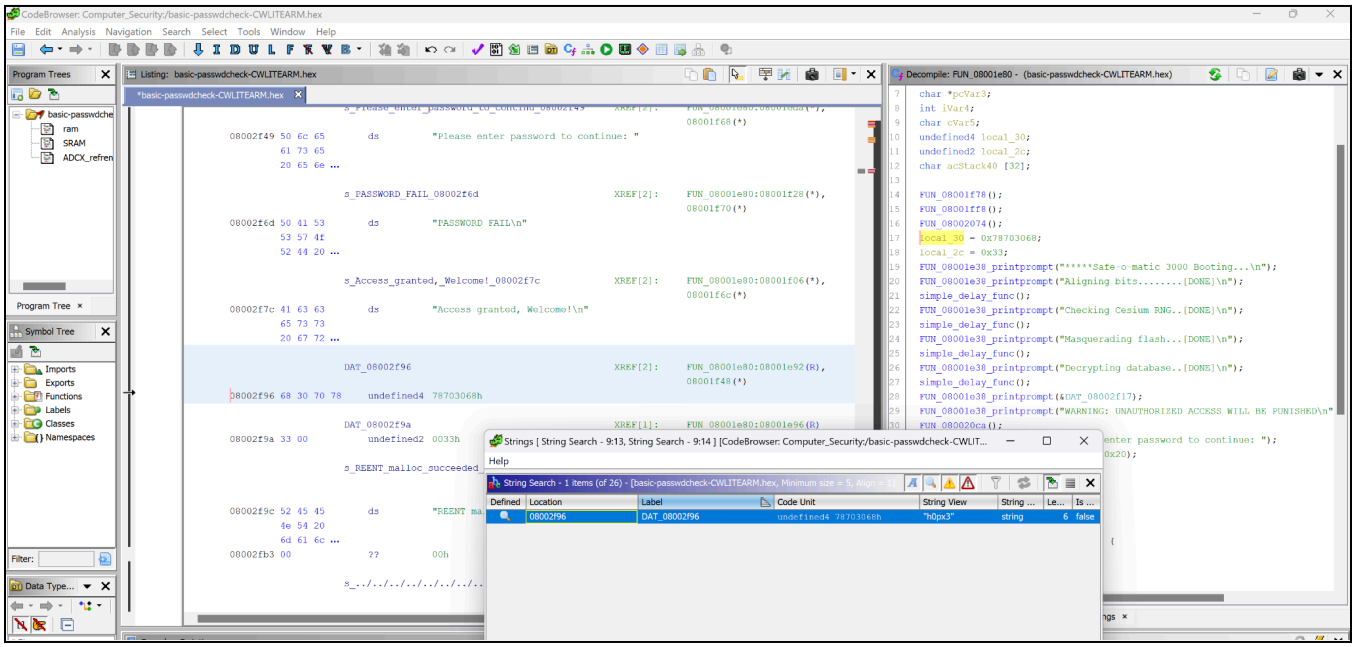
It seems like the password in our case is hardcoded to be: local_30 = 0x78703068; which can be converted to “xp0h” with big endian or “h0px” with little endian, it is then appended with local 2c making it “**h0px3**”.

After a bit more digging around I found the following password too! I wonder if this is for another lab or another purpose?!

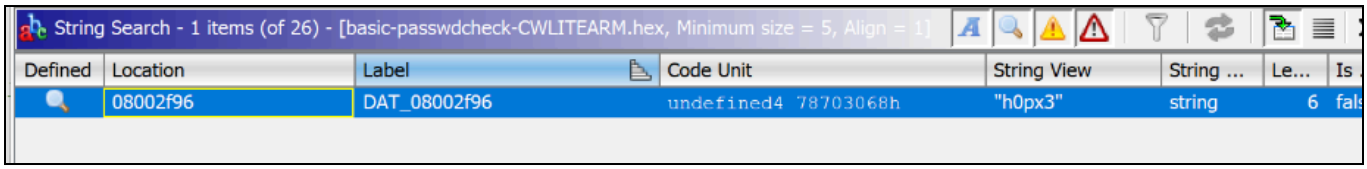
```
080007da 60 48      ldr       r0=>s_efgEFG_08003034,[PTR_s_efgEFG_0800095c]    = 08003034
```

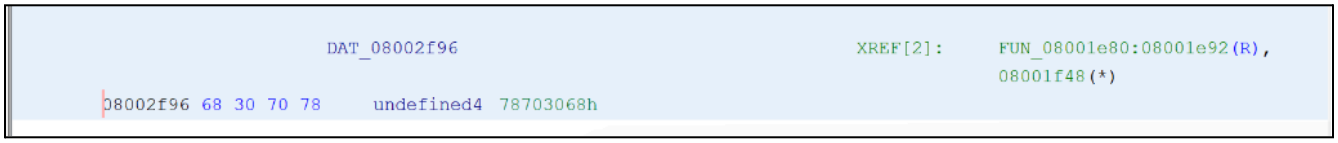
= "efgEFG"

c. Where is the password stored? [1 pts]



Zooming in:





It is stored in 0x0800 2f96 in memory.

Conclusion

In conclusion, this lab has given us a successful trial of ghidra introducing us hands on to the concept of software reverse engineering and allowing us to use it first hand to find useful data and find the password used in this program. We configured tools, analyzed segments and explored binary structure, memory maps and peripheral usage and set up. This has greatly enhanced our skills in computer security.