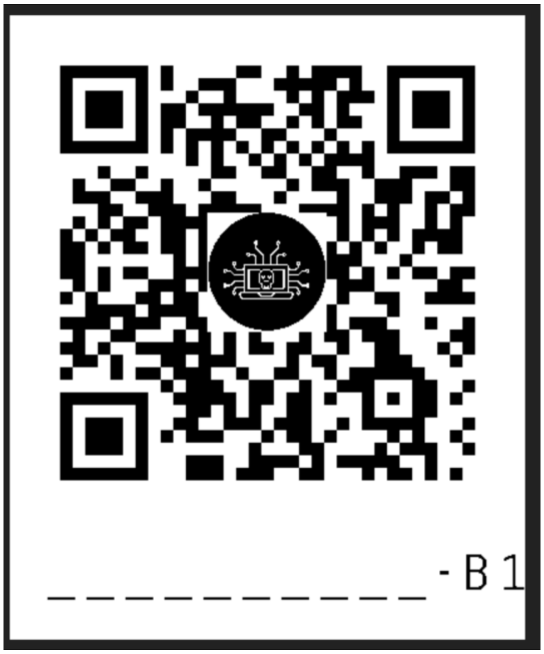
HW2

<u>Full name 1:</u> Roni Roitbord
<u>ID1</u>: 313575599

<u>Full name 2:</u> Davis Birman Tobias
<u>ID1</u>: 342662798

Wizard

We began at downloading all the files from the files tab as the wizard user and as we noticed that he lost 335/351 games we wanted to change the user we chose but time is precious. We used analyzer.exe on qrcode.png and received the following message:
“Image contains something hidden in the background. Check this out.”
with the following image:

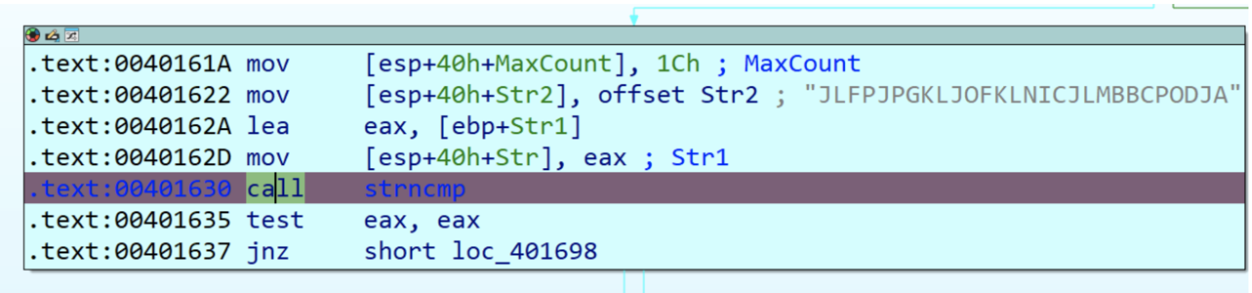


We also notice that on HW1 we found 4 credentials to the site but according to save.pdf (on the public files) only 3 users we're involved (as for now).

By running ida with safe.exe, and checking the strings (view->subviews->strings) we noticed a suspicious string:

JLFPJPGKLJOFKLNICJLMBBCPODJA

After debugging with IDA and searching for relevant code flows which used scanf we noticed this particularly block:



```
.text:0040161A mov     [esp+40h+MaxCount], 1Ch ; MaxCount
.text:00401622 mov     [esp+40h+Str2], offset Str2 ; "JLFPJPGKLJOFKLNICJLMBBCPODJA"
.text:0040162A lea     eax, [ebp+Str1]
.text:0040162D mov     [esp+40h+Str], eax ; Str1
.text:00401630 call    strncmp
.text:00401635 test    eax, eax
.text:00401637 jnz     short loc_401698
```

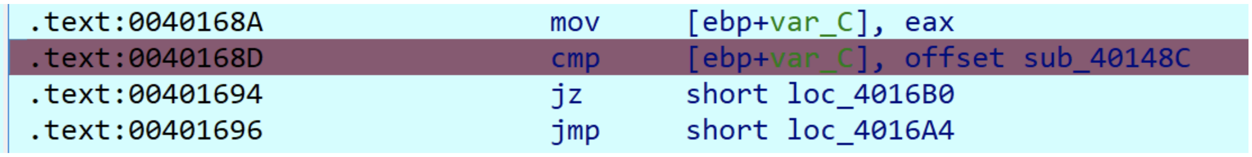
Observing it and the code flow above it, we understand that ebp+Str1 had a permutation of the input, so we need to find the right input which will be compared to the string for **the first 28 characters**. After investigating the code, we understood that the input was modified by the following algorithm:

1. Subtract 65 from each char.
2. XOR each number with the next one (wrap the last char to the first one)
3. add 65 to each char.

According to this algorithm, we managed to reverse it and find the following string: DKBELCNLBKDNICJEMOHMABACNDAJJ

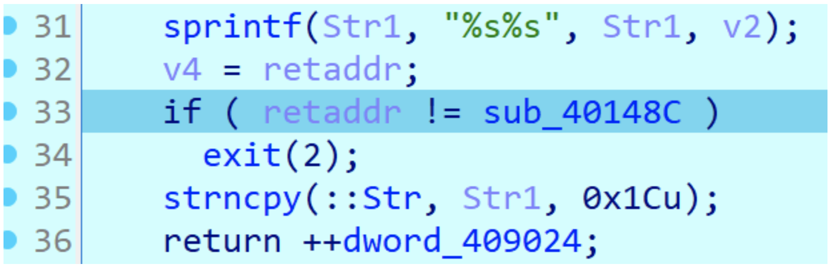
Then we passed the previous block (and not to the exit block) due to the strncmp function.

Next, we encountered the following condition:



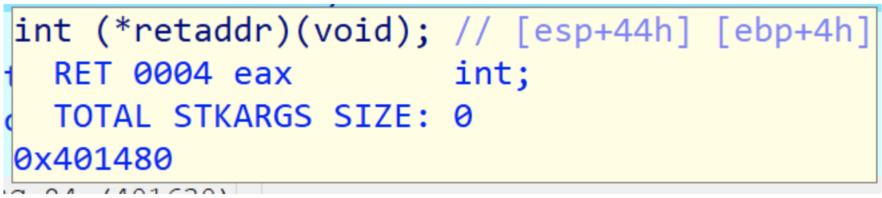
```
.text:0040168A                 mov     [ebp+var_C], eax
.text:0040168D                 cmp     [ebp+var_C], offset sub_40148C
.text:00401694                 jz      short loc_4016B0
.text:00401696                 jmp     short loc_4016A4
```

After checking the pseudo code of this condition:



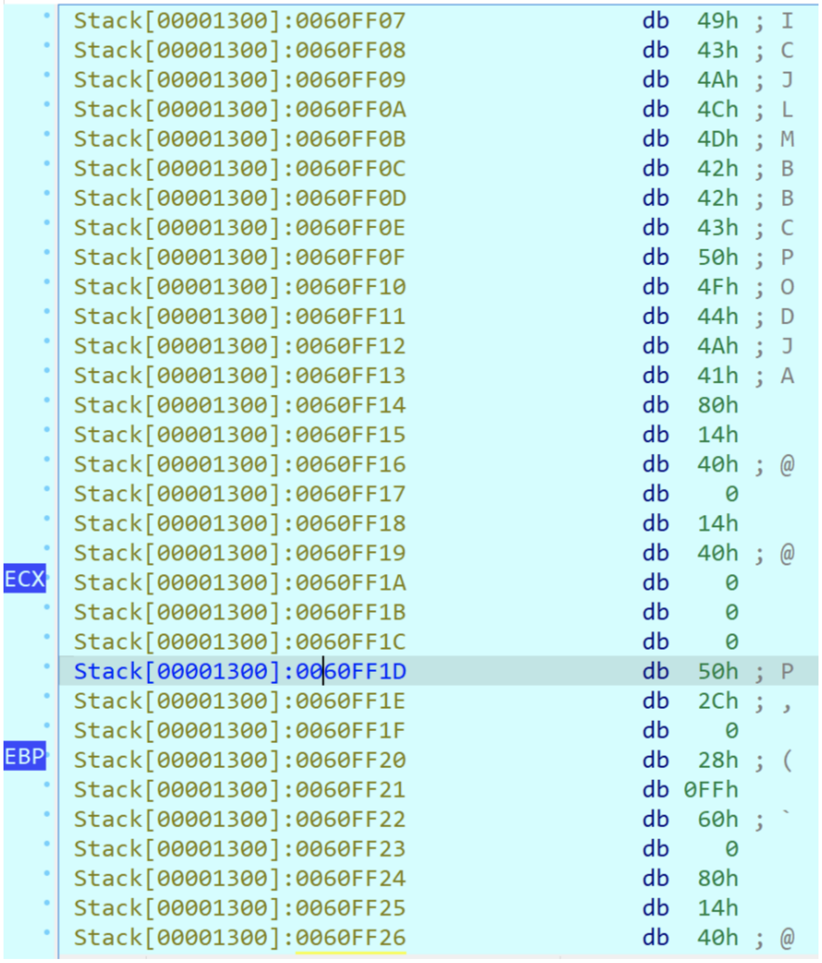
```
  sprintf(Str1, "%s%s", Str1, v2);
  v4 = retaddr;
  if ( retaddr != sub_40148C )
    exit(2);
  strncpy(::Str, Str1, 0x1Cu);
  return ++dword_409024;
```

We realized that we need to adjust our return address to 0x40148C. by checking the return address:



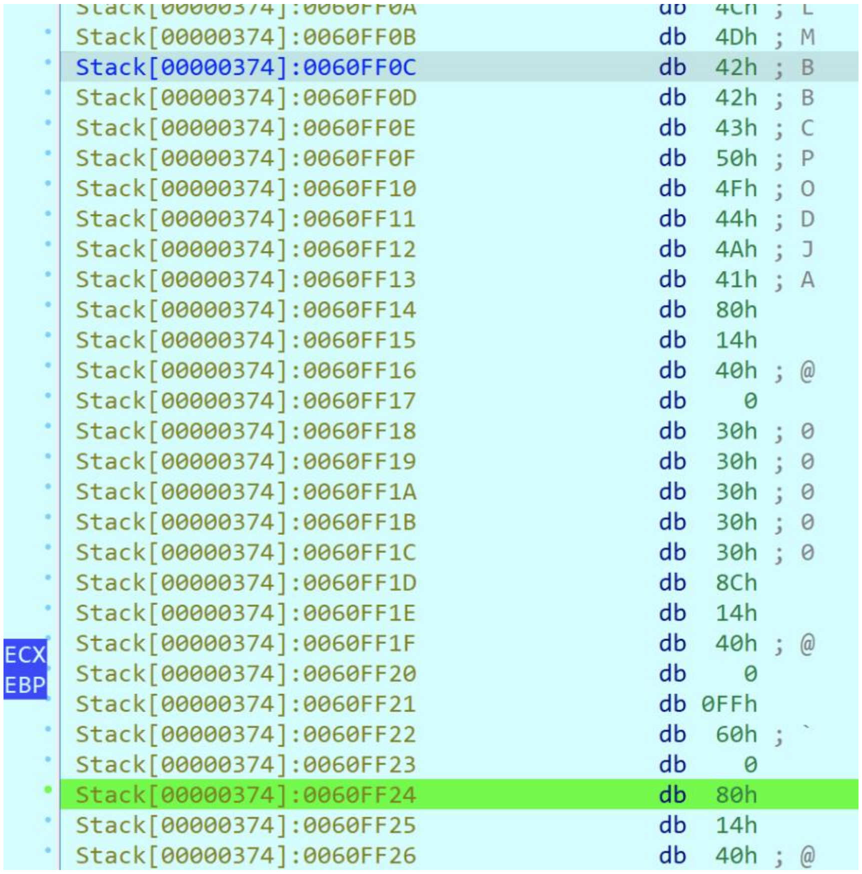
```
int (*retaddr)(void); // [esp+44h] [ebp+4h]
  RET 0004 eax          int;
  TOTAL STKARGS SIZE: 0
0x401480
```

We decided to check our stack:



```
    Stack[00001300]:0060FF07                db  49h ; I
    Stack[00001300]:0060FF08                db  43h ; C
    Stack[00001300]:0060FF09                db  4Ah ; J
    Stack[00001300]:0060FF0A                db  4Ch ; L
    Stack[00001300]:0060FF0B                db  4Dh ; M
    Stack[00001300]:0060FF0C                db  42h ; B
    Stack[00001300]:0060FF0D                db  42h ; B
    Stack[00001300]:0060FF0E                db  43h ; C
    Stack[00001300]:0060FF0F                db  50h ; P
    Stack[00001300]:0060FF10                db  4Fh ; O
    Stack[00001300]:0060FF11                db  44h ; D
    Stack[00001300]:0060FF12                db  4Ah ; J
    Stack[00001300]:0060FF13                db  41h ; A
    Stack[00001300]:0060FF14                db  80h
    Stack[00001300]:0060FF15                db  14h
    Stack[00001300]:0060FF16                db  40h ; @
    Stack[00001300]:0060FF17                db    0
    Stack[00001300]:0060FF18                db  14h
    Stack[00001300]:0060FF19                db  40h ; @
ECX Stack[00001300]:0060FF1A                db    0
    Stack[00001300]:0060FF1B                db    0
    Stack[00001300]:0060FF1C                db    0
    Stack[00001300]:0060FF1D                db  50h ; P
    Stack[00001300]:0060FF1E                db  2Ch ; ,
    Stack[00001300]:0060FF1F                db    0
EBP Stack[00001300]:0060FF20                db  28h ; (
    Stack[00001300]:0060FF21                db 0FFh
    Stack[00001300]:0060FF22                db  60h ; `
    Stack[00001300]:0060FF23                db    0
    Stack[00001300]:0060FF24                db  80h
    Stack[00001300]:0060FF25                db  14h
    Stack[00001300]:0060FF26                db  40h ; @
```

ebp+4 holds the value of 0x401480 but it doesn’t help us as we need to push the values from within the stack and then we remembered that the first condition was comparing the first 28 characters so we could write a longer string and it will still pass the first condition and then we noticed that when we push more characters to our input, we can see the right value for our second condition:

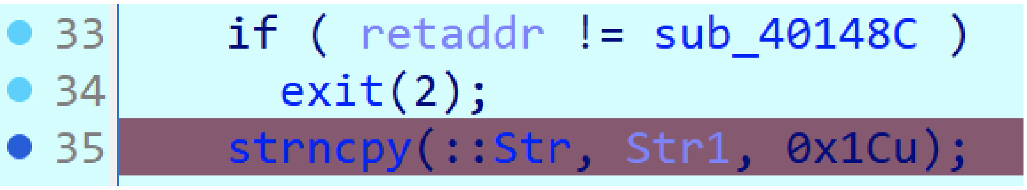


```
    Stack[00000374]:0060FF0A          db  4Ch ; L
    Stack[00000374]:0060FF0B          db  4Dh ; M
    Stack[00000374]:0060FF0C          db  42h ; B
    Stack[00000374]:0060FF0D          db  42h ; B
    Stack[00000374]:0060FF0E          db  43h ; C
    Stack[00000374]:0060FF0F          db  50h ; P
    Stack[00000374]:0060FF10          db  4Fh ; O
    Stack[00000374]:0060FF11          db  44h ; D
    Stack[00000374]:0060FF12          db  4Ah ; J
    Stack[00000374]:0060FF13          db  41h ; A
    Stack[00000374]:0060FF14          db  80h
    Stack[00000374]:0060FF15          db  14h
    Stack[00000374]:0060FF16          db  40h ; @
    Stack[00000374]:0060FF17          db    0
    Stack[00000374]:0060FF18          db  30h ; 0
    Stack[00000374]:0060FF19          db  30h ; 0
    Stack[00000374]:0060FF1A          db  30h ; 0
    Stack[00000374]:0060FF1B          db  30h ; 0
    Stack[00000374]:0060FF1C          db  30h ; 0
    Stack[00000374]:0060FF1D          db  8Ch
    Stack[00000374]:0060FF1E          db  14h
ECX Stack[00000374]:0060FF1F          db  40h ; @
EBP Stack[00000374]:0060FF20          db    0
    Stack[00000374]:0060FF21          db 0FFh
    Stack[00000374]:0060FF22          db  60h ; `
    Stack[00000374]:0060FF23          db    0
    Stack[00000374]:0060FF24          db  80h
    Stack[00000374]:0060FF25          db  14h
    Stack[00000374]:0060FF26          db  40h ; @
```

We count how much additional characters we need to provide in order to push the right values into ebp+4 and it’s 15 additional characters so we used the following string:

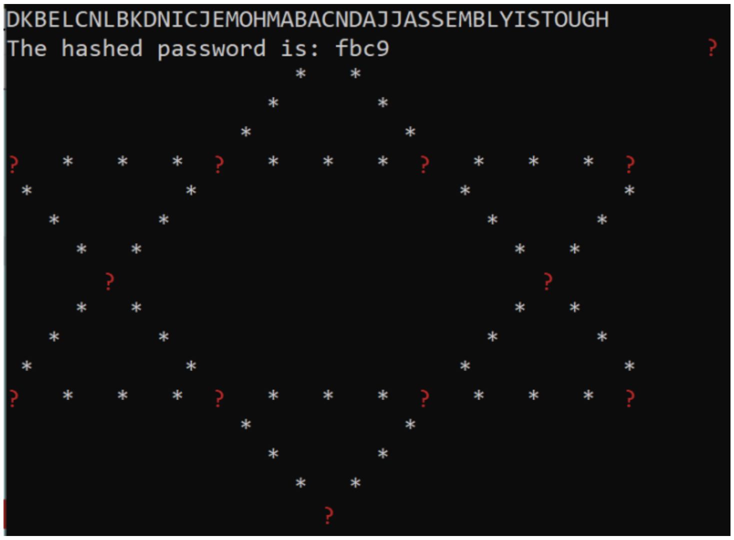
DKBELCNLBKDNICJEMOHMABACNDAJJASSEMBLYISTOUGH

And we passed the second condition:



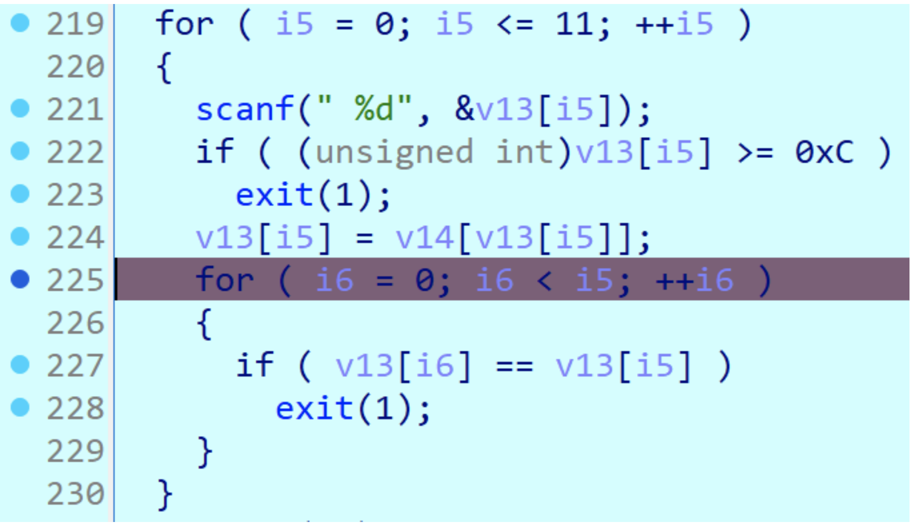
```
  if ( retaddr != sub_40148C )
    exit(2);
  strncpy(::Str, Str1, 0x1Cu);
```

Running the new input provides the following star of David and a hashed password **fbc9**:



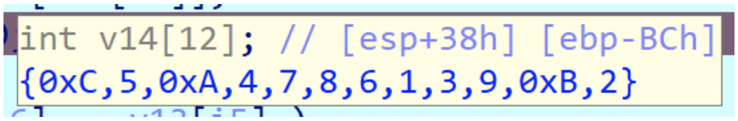
```
DKBELCNLBKDNICJEMOHMABACNDAJJASSEMBLYISTOUGH
The hashed password is: fbc9                       ?
                     *   *
                   *       *
                 *           *
?   *   *   *  ?   *   *   *  ?   *   *   *  ?
 *           *                   *           *
   *       *                       *       *
     *   *                           *   *
       ?                               ?
     *   *                           *   *
   *       *                       *       *
 *           *                   *           *
?   *   *   *  ?   *   *   *  ?   *   *   *  ?
                 *           *
                   *       *
                     *   *
                       ?
```

Now the program waits for another input, so we search for the next scanf and we notice that the expected input is a number (due to the %d) we convert the code into a pseudo code:



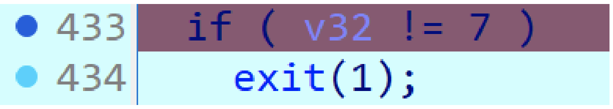
```
for ( i5 = 0; i5 <= 11; ++i5 )
{
  scanf(" %d", &v13[i5]);
  if ( (unsigned int)v13[i5] >= 0xC )
    exit(1);
  v13[i5] = v14[v13[i5]];
  for ( i6 = 0; i6 < i5; ++i6 )
  {
    if ( v13[i6] == v13[i5] )
      exit(1);
  }
}
```

We observed the following: a loop of 12 iterations, each input should be equal or lower to 0xC (12), each input should be different from another, to summarize we need to reorder the first 12 numbers ($0 \leq x \leq 11$) with respect to another variable (which is shown as v14) so for that we need to understand what is v14:



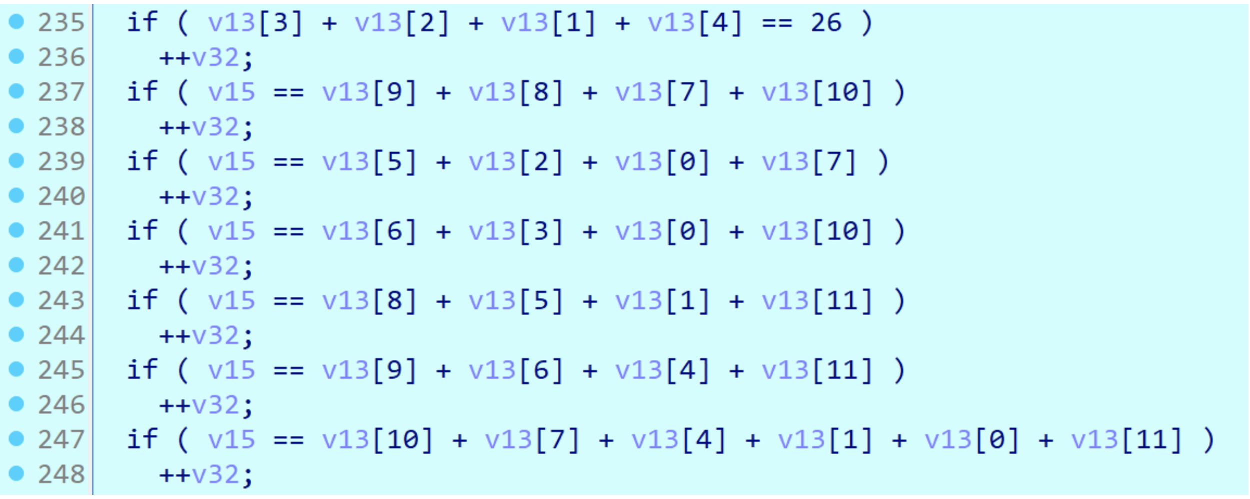
```
int v14[12]; // [esp+38h] [ebp-BCh]
{0xC,5,0xA,4,7,8,6,1,3,9,0xB,2}
```

It seems that we need to input permutation of $\{0, \dots, 11\}$, every such permutation will print a different star of David (with numbers instead of question marks) but then exit. Checking the prior condition will give us more insight:



```
if ( v32 != 7 )
  exit(1);
```

And when checking for incrementation of v32, we found these equations:

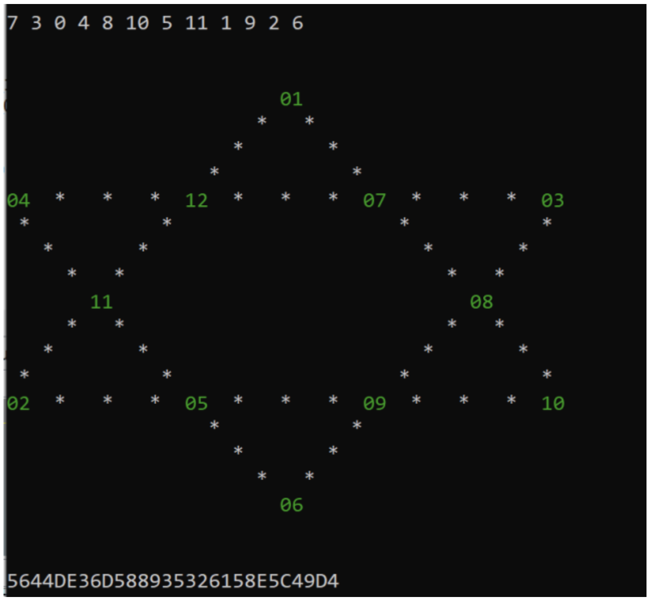


```
if ( v13[3] + v13[2] + v13[1] + v13[4] == 26 )
  ++v32;
if ( v15 == v13[9] + v13[8] + v13[7] + v13[10] )
  ++v32;
if ( v15 == v13[5] + v13[2] + v13[0] + v13[7] )
  ++v32;
if ( v15 == v13[6] + v13[3] + v13[0] + v13[10] )
  ++v32;
if ( v15 == v13[8] + v13[5] + v13[1] + v13[11] )
  ++v32;
if ( v15 == v13[9] + v13[6] + v13[4] + v13[11] )
  ++v32;
if ( v15 == v13[10] + v13[7] + v13[4] + v13[1] + v13[0] + v13[11] )
  ++v32;
```

In order to solve these equations, we used unconventional ways due to a couple reasons:

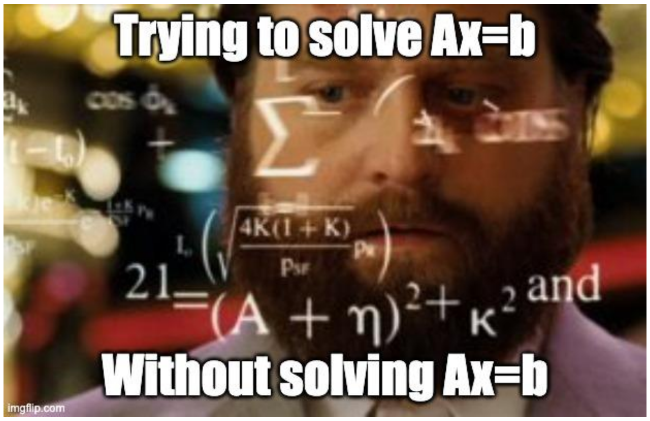
- Brute force is not an option (12! Permutations)
- First 2 equations include different indexes
- There are a limited set of groups of 2 sub-groups sized 4, with different 8 numbers with the restriction $x_i \in [1,12]$ which drastically lowering our solution area from 12! to $4! \cdot 4! \cdot 2 \cdot 6!$

To find the solution, we used *StarOfDavid.py* file to find all possible solutions and then map it to v14 variable (as we showed above), we chose the solution
1, 4, 12, 7, 3, 11, 8, 2, 5, 9, 10, 6
so according to the mapping, our input will be:
7 3 0 4 8 10 5 11 1 9 2 6



```
7 3 0 4 8 10 5 11 1 9 2 6


                       01
                     *   *
                   *       *
                 *           *
04  *   *   *  12  *   *   *  07  *   *   *  03
 *           *                   *           *
   *       *                       *       *
     *   *                           *   *
       11                              08
     *   *                           *   *
   *       *                       *       *
 *           *                   *           *
02  *   *   *  05  *   *   *  09  *   *   *  10
                 *           *
                   *       *
                     *   *
                       06


5644DE36D588935326158E5C49D4
```

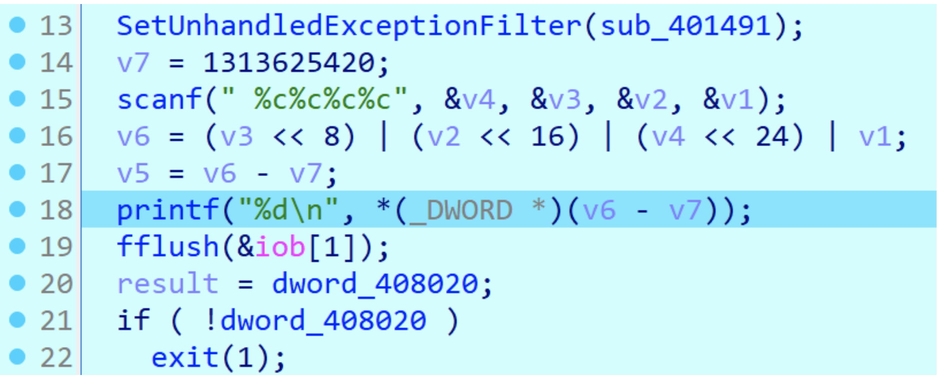
The vertices are green! And we received the following string: 5644DE36D588935326158E5C49D4



According to save.pdf, we’ve found our encrypted password! Let’s continue to our giant’s encryption parameters.

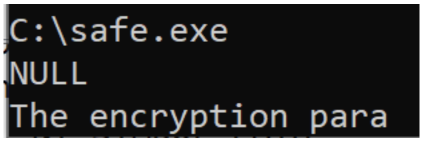
Giant

The same as with the wizard, we downloaded the giant's files and investigated the safe.exe file with IDA, as we executed the file it waited for an input and after we provided some random inputs it exited (like with the magician's safe.exe) so we checked for scanf calls and we've found 4 calls. We checked the first one and the input it expected to receive:



```
SetUnhandledExceptionFilter(sub_401491);
v7 = 1313625420;
scanf(" %c%c%c%c", &v4, &v3, &v2, &v1);
v6 = (v3 << 8) | (v2 << 16) | (v4 << 24) | v1;
v5 = v6 - v7;
printf("%d\n", *(_DWORD *)(v6 - v7));
fflush(&iob[1]);
result = dword_408020;
if ( !dword_408020 )
  exit(1);
```

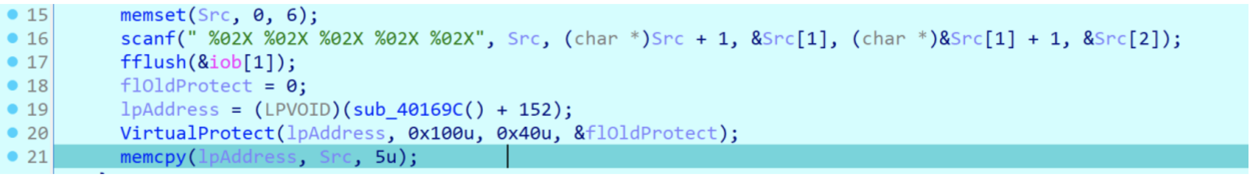
We had an exception while referencing to the memory stored on v5 so in order to find the right value, we noticed that scanf expects for 4 characters and the ascii value of v7 is NLUL which may imply that the first input should be NULL, we tested it and got passed our exception:



```
C:\safe.exe
NULL
The encryption para
```

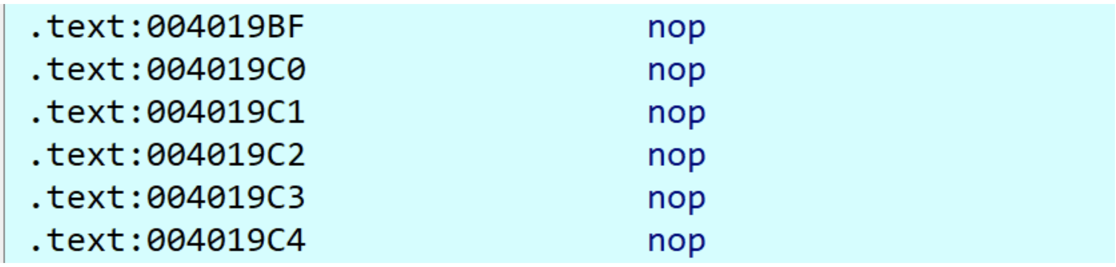
In order to progress with debugging, we set the ip on the first command in the next block.

Checking the second scanf, the expected input is 5 unsigned hex values with at least 2 characters each (and pad with 0 if provided less than 2 characters for each input):



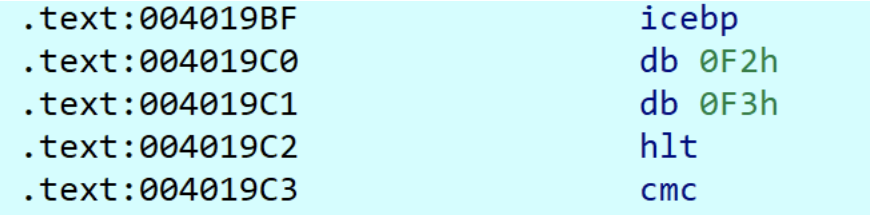
```
memset(Src, 0, 6);
scanf(" %02X %02X %02X %02X %02X", Src, (char *)Src + 1, &Src[1], (char *)&Src[1] + 1, &Src[2]);
fflush(&iob[1]);
flOldProtect = 0;
lpAddress = (LPVOID)(sub_40169C() + 152);
VirtualProtect(lpAddress, 0x100u, 0x40u, &flOldProtect);
memcpy(lpAddress, Src, 5u);
```

After couple of tests, we noticed that our input modifies some of the instructions, before random input:



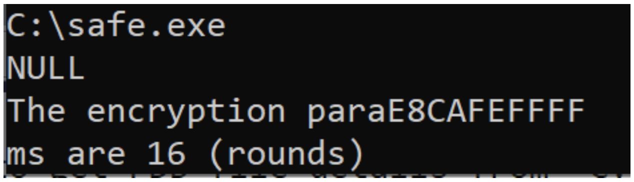
```
.text:004019BF                  nop
.text:004019C0                  nop
.text:004019C1                  nop
.text:004019C2                  nop
.text:004019C3                  nop
.text:004019C4                  nop
```

after random input:



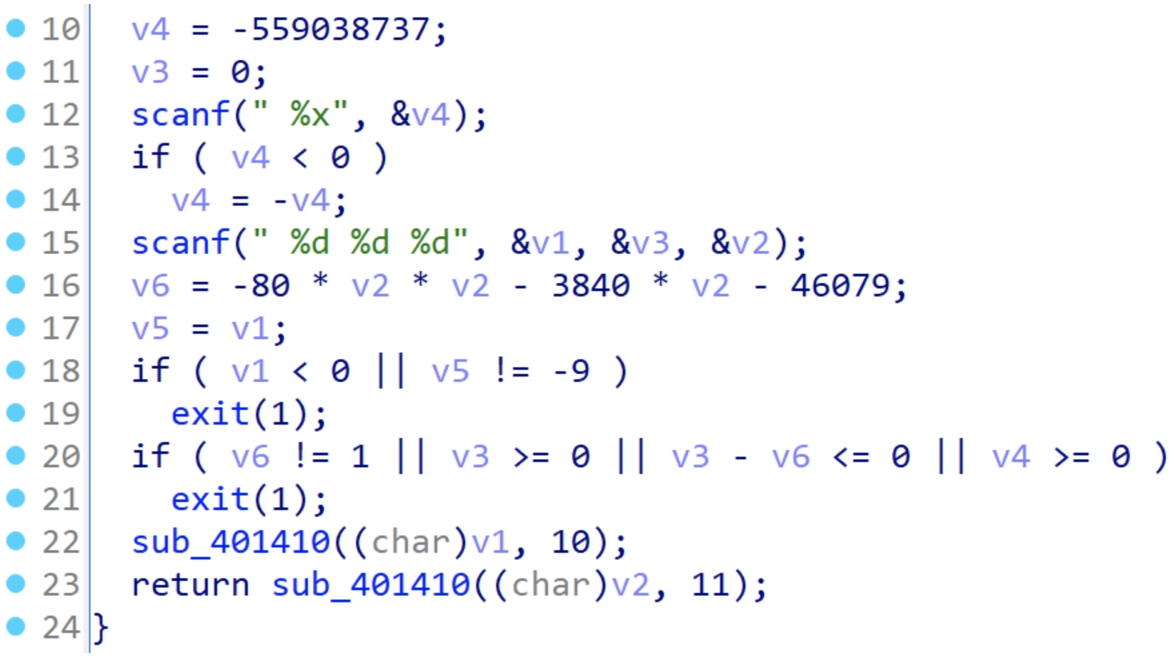
```
.text:004019BF                  icebp
.text:004019C0                  db 0F2h
.text:004019C1                  db 0F3h
.text:004019C2                  hlt
.text:004019C3                  cmc
```

After long investigation, we found that there is a recursive function which swaps between the input values in the stack, with the example of part of the function and the similarity between that block and the one which filled with nops, we found that the recursion part is missing from the second part so we decided to call itself with E8 instruction and 4 bytes which will be the distance between the beginning of the function and the missing code and we found that the right input is E8 CA FE FF FF which provided us the next input:



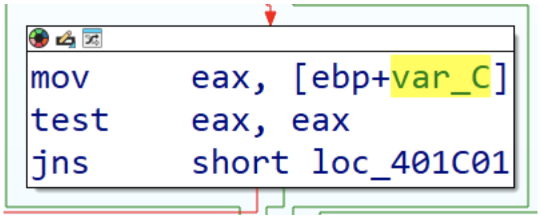
```
C:\safe.exe
NULL
The encryption paraE8CAFEFFFF
ms are 16 (rounds)
```

Moving to the third scanf, which receives a hex value:



```
10  v4 = -559038737;
11  v3 = 0;
12  scanf(" %x", &v4);
13  if ( v4 < 0 )
14    v4 = -v4;
15  scanf(" %d %d %d", &v1, &v3, &v2);
16  v6 = -80 * v2 * v2 - 3840 * v2 - 46079;
17  v5 = v1;
18  if ( v1 < 0 || v5 != -9 )
19    exit(1);
20  if ( v6 != 1 || v3 >= 0 || v3 - v6 <= 0 || v4 >= 0 )
21    exit(1);
22  sub_401410((char)v1, 10);
23  return sub_401410((char)v2, 11);
24 }
```

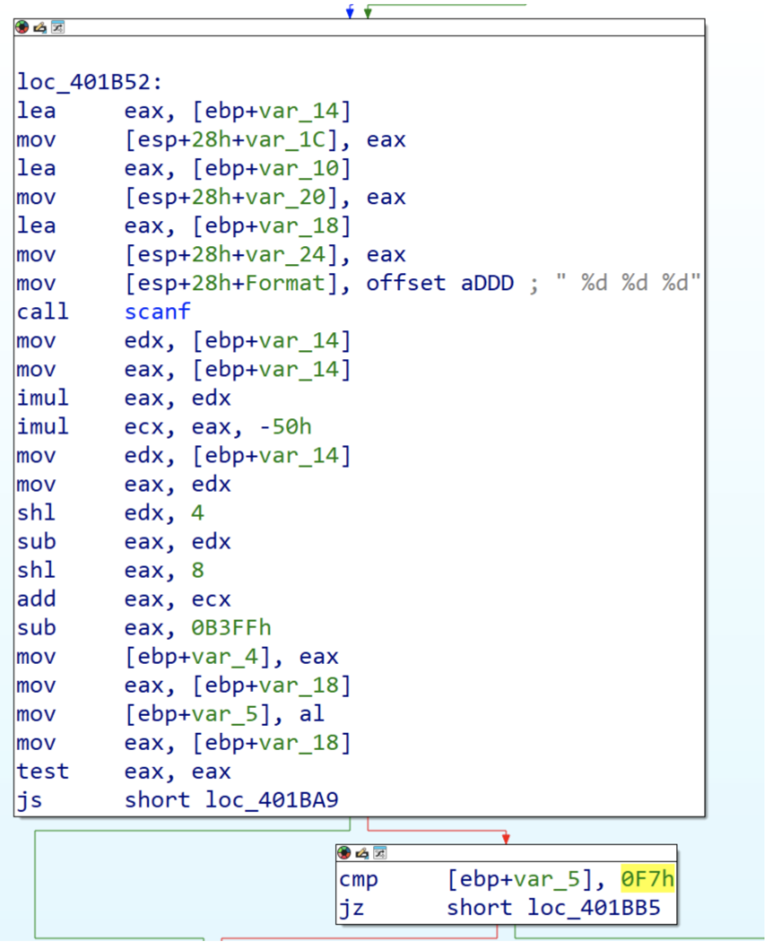
Here we can see that there's a strange conditioning: according to lines 13 & 14, if v4 is negative, convert it to non-negative and on line 20, if v4 is non-negative, exit the program. This was a tricky one but eventually we noticed how this condition was created:



```
mov     eax, [ebp+var_C]
test    eax, eax
jns     short loc_401C01
```

In order to pass that we need to find a large number with MSB of 1 so the SF flag will be set and the jump instruction won't occur so we can use the smallest negative number on x32 which is 80000000

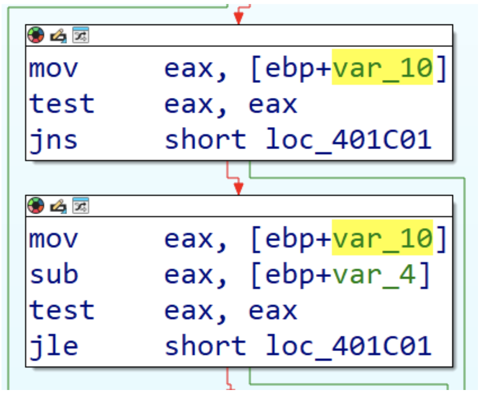
Next, we need to find the right values for the 4th and last scanf, which accepts 3 integers:



```
loc_401B52:
lea     eax, [ebp+var_14]
mov     [esp+28h+var_1C], eax
lea     eax, [ebp+var_10]
mov     [esp+28h+var_20], eax
lea     eax, [ebp+var_18]
mov     [esp+28h+var_24], eax
mov     [esp+28h+Format], offset aDDD ; " %d %d %d"
call    scanf
mov     edx, [ebp+var_14]
mov     eax, [ebp+var_14]
imul    eax, edx
imul    ecx, eax, -50h
mov     edx, [ebp+var_14]
mov     eax, edx
shl     edx, 4
sub     eax, edx
shl     eax, 8
add     eax, ecx
sub     eax, 0B3FFh
mov     [ebp+var_4], eax
mov     eax, [ebp+var_18]
mov     [ebp+var_5], al
mov     eax, [ebp+var_18]
test    eax, eax
js      short loc_401BA9

cmp     [ebp+var_5], 0F7h
jz      short loc_401BB5
```

We can see that all the variables stored in eax, and the first byte should be the decimal value of F7, which means it's 247.

The second variable should be non-negative but if you subtract from it the solution of the equation it should be non-positive, the solution of the equation should be 1 (part of the conditions) so $x \geq 0 \,\&\, x - 1 \leq 0 \rightarrow x \in \{0,1\}$ but comparing it to the original code provided a bit different solution:



```
mov     eax, [ebp+var_10]
test    eax, eax
jns     short loc_401C01

mov     eax, [ebp+var_10]
sub     eax, [ebp+var_4]
test    eax, eax
jle     short loc_401C01
```

as we tried to follow the code follow and noticed another jns, we tried to cause integer overflow by using -2147483648 as the second variable

Next, we needed to solve the following equation for the last input:

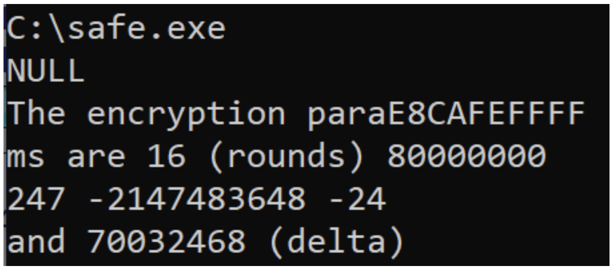
$$-80x^2 - 3840x - 46079 = 1$$

$$-80x^2 - 3840x - 46080 = 0$$

$$x^2 + 48x + 576 = 0 \rightarrow x = -24$$

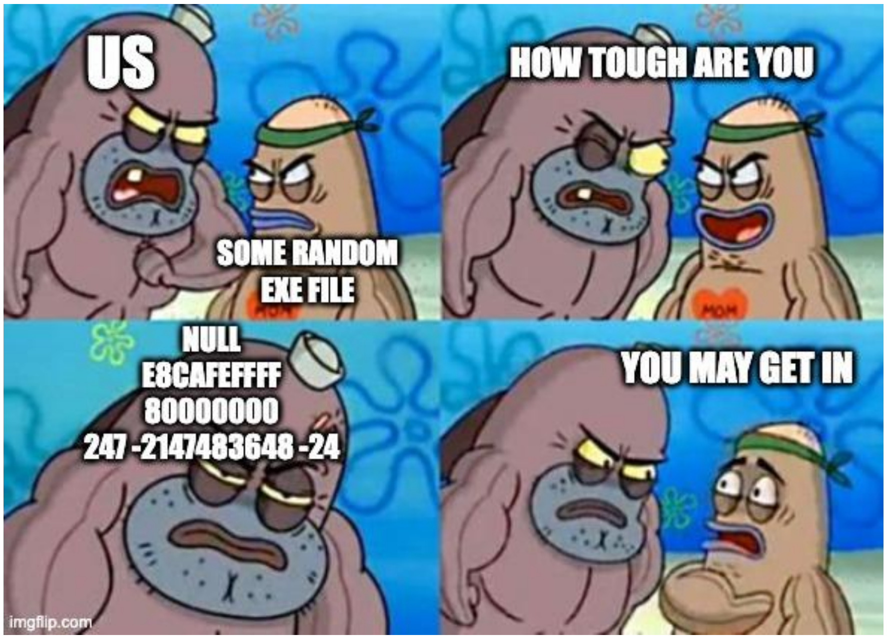
For summary, the inputs we've found are:

80000000
247 -2147483648 -24



```
C:\safe.exe
NULL
The encryption paraE8CAFEFFFF
ms are 16 (rounds) 80000000
247 -2147483648 -24
and 70032468 (delta)
```

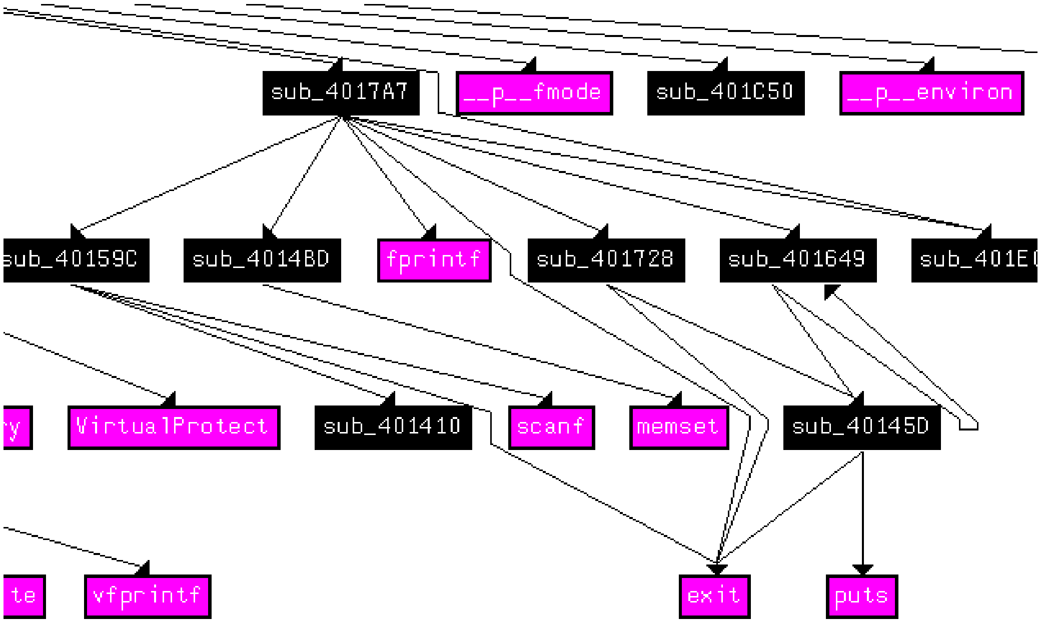
After cleaning the prompt input, we received the following output:
The encryption params are 16 (rounds) and 70032468 (delta)



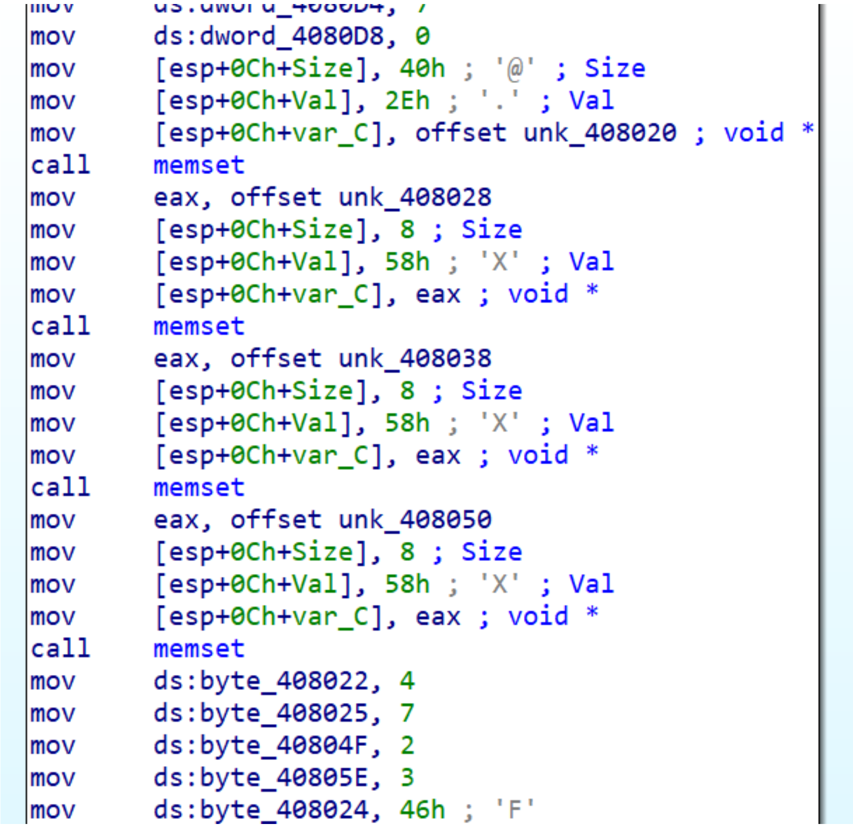
Gobling

Firstly, we started trying to analyze everything and got pretty afraid of the tlscallback_0 after wee read on google that it could be an anti-debugging technique, so a lot of time was spent trying to figure it out if the cpuid instructions could realize we are debugging and change the code on the flow.

We used ida to give us a graph of functions calls, so we could locate possible memory changing and input/output calls.



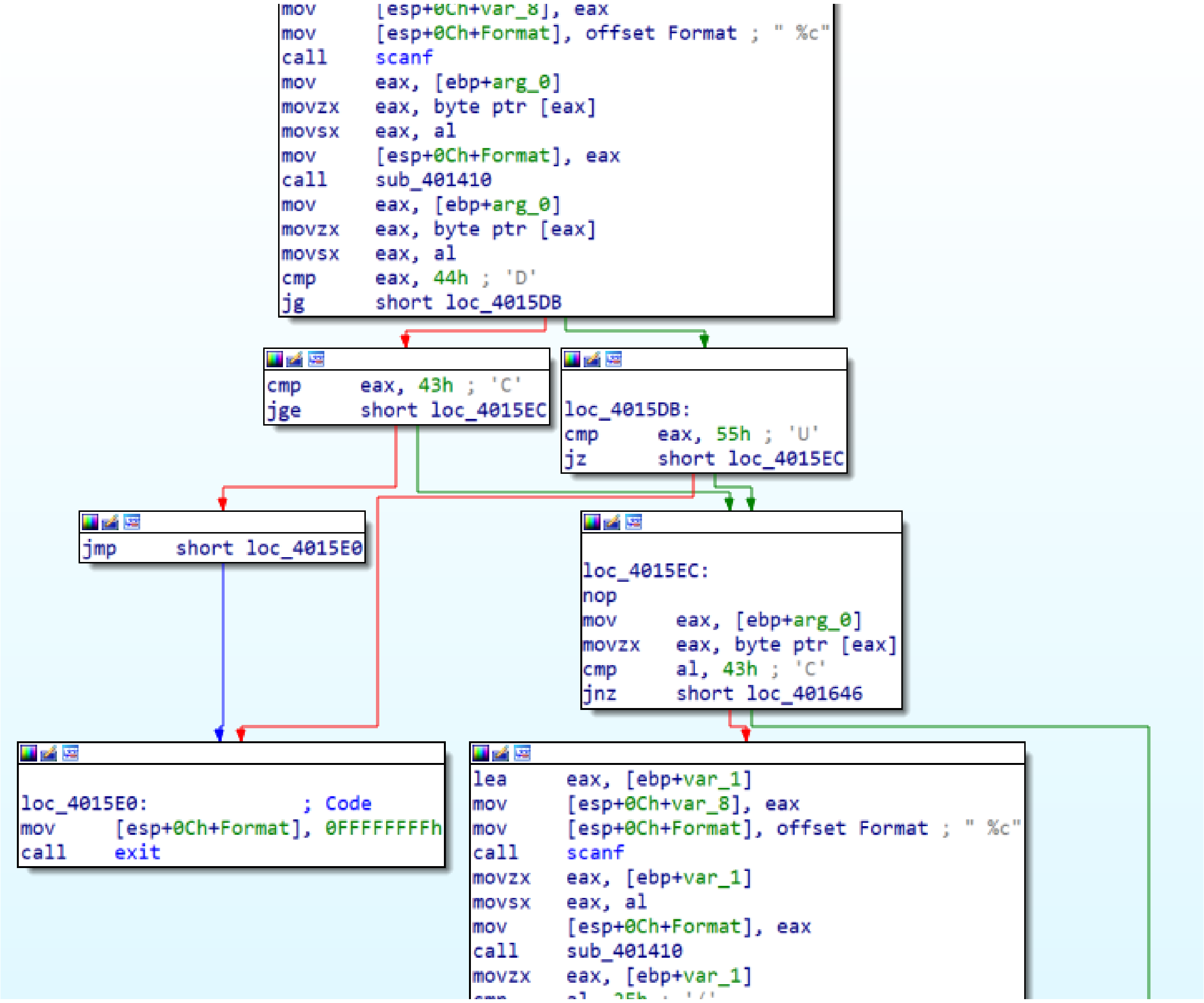
We could not find any indication of code changing on the fly so we figured out that we should focus our attention on the input and output of safe.exe. While looking for memory changing, we stumbled upon this code:



```
mov     ds:dword_4080D4, 7
mov     ds:dword_4080D8, 0
mov     [esp+0Ch+Size], 40h ; '@' ; Size
mov     [esp+0Ch+Val], 2Eh ; '.' ; Val
mov     [esp+0Ch+var_C], offset unk_408020 ; void *
call    memset
mov     eax, offset unk_408028
mov     [esp+0Ch+Size], 8 ; Size
mov     [esp+0Ch+Val], 58h ; 'X' ; Val
mov     [esp+0Ch+var_C], eax ; void *
call    memset
mov     eax, offset unk_408038
mov     [esp+0Ch+Size], 8 ; Size
mov     [esp+0Ch+Val], 58h ; 'X' ; Val
mov     [esp+0Ch+var_C], eax ; void *
call    memset
mov     eax, offset unk_408050
mov     [esp+0Ch+Size], 8 ; Size
mov     [esp+0Ch+Val], 58h ; 'X' ; Val
mov     [esp+0Ch+var_C], eax ; void *
call    memset
mov     ds:byte_408022, 4
mov     ds:byte_408025, 7
mov     ds:byte_40804F, 2
mov     ds:byte_40805E, 3
mov     ds:byte_408024, 46h ; 'F'
```

We did not know what it meant but clearly represented some initialization of memory, that presented very useful as we proceeded the analysis.

We started looking at the legal inputs for the executable and we found



```
mov     [esp+0Ch+var_8], eax
mov     [esp+0Ch+Format], offset Format ; " %c"
call    scanf
mov     eax, [ebp+arg_0]
movzx   eax, byte ptr [eax]
movsx   eax, al
mov     [esp+0Ch+Format], eax
call    sub_401410
mov     eax, [ebp+arg_0]
movzx   eax, byte ptr [eax]
movsx   eax, al
cmp     eax, 44h ; 'D'
jg      short loc_4015DB
```

```
cmp     eax, 43h ; 'C'
jge     short loc_4015EC
```

```
loc_4015DB:
cmp     eax, 55h ; 'U'
jz      short loc_4015EC
```

```
jmp     short loc_4015E0
```

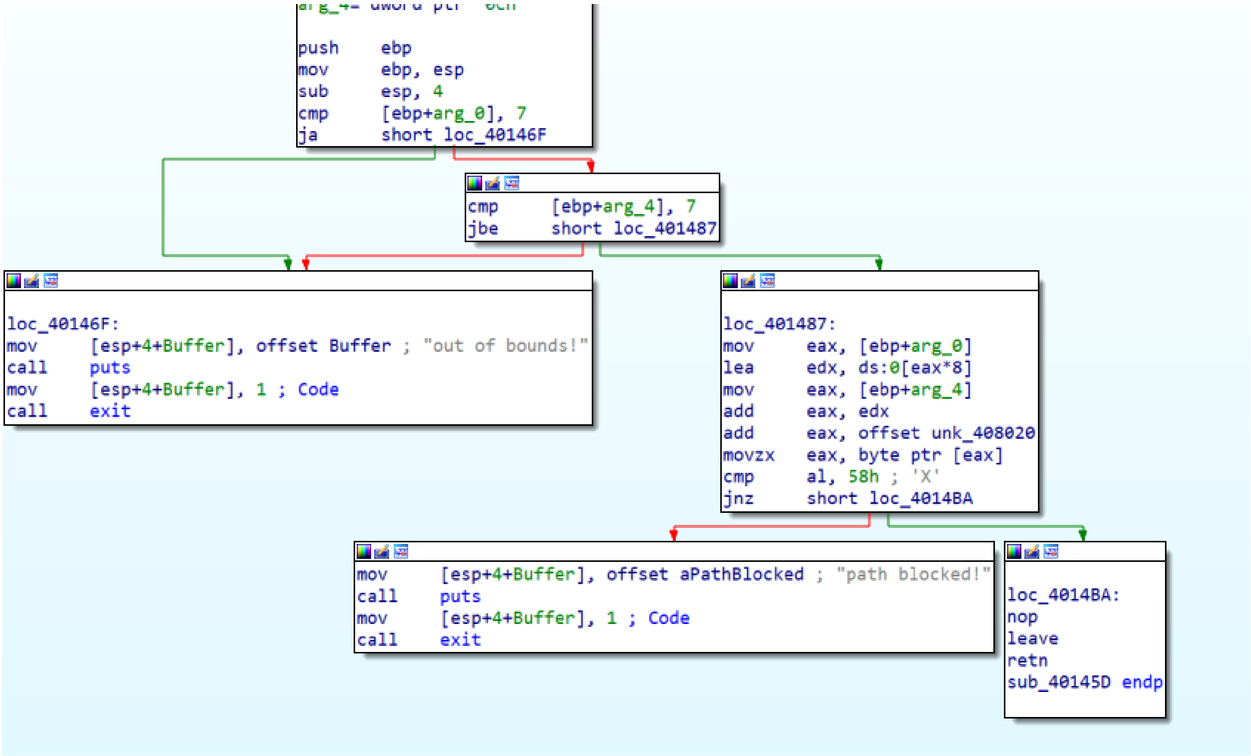
```
loc_4015EC:
nop
mov     eax, [ebp+arg_0]
movzx   eax, byte ptr [eax]
cmp     al, 43h ; 'C'
jnz     short loc_401646
```

```
loc_4015E0:                     ; Code
mov     [esp+0Ch+Format], 0FFFFFFFFh
call    exit
```

```
lea     eax, [ebp+var_1]
mov     [esp+0Ch+var_8], eax
mov     [esp+0Ch+Format], offset Format ; " %c"
call    scanf
movzx   eax, [ebp+var_1]
movsx   eax, al
mov     [esp+0Ch+Format], eax
call    sub_401410
movzx   eax, [ebp+var_1]
cmp     al, 2Fh ; '/'
```

It took us some time to understand it, but we figured that legal input was either ‘U’, ’D’ or ‘C#’ where ‘#’ represents a digit.

Then we started looking at where we had output. There we some error messages that at first, we ignored (and ultimately were indeed not relevant). And then we found



```
arg_4= dword ptr  0Ch

push    ebp
mov     ebp, esp
sub     esp, 4
cmp     [ebp+arg_0], 7
ja      short loc_40146F
```

```
cmp     [ebp+arg_4], 7
jbe     short loc_401487
```

```
loc_40146F:
mov     [esp+4+Buffer], offset Buffer ; "out of bounds!"
call    puts
mov     [esp+4+Buffer], 1 ; Code
call    exit
```

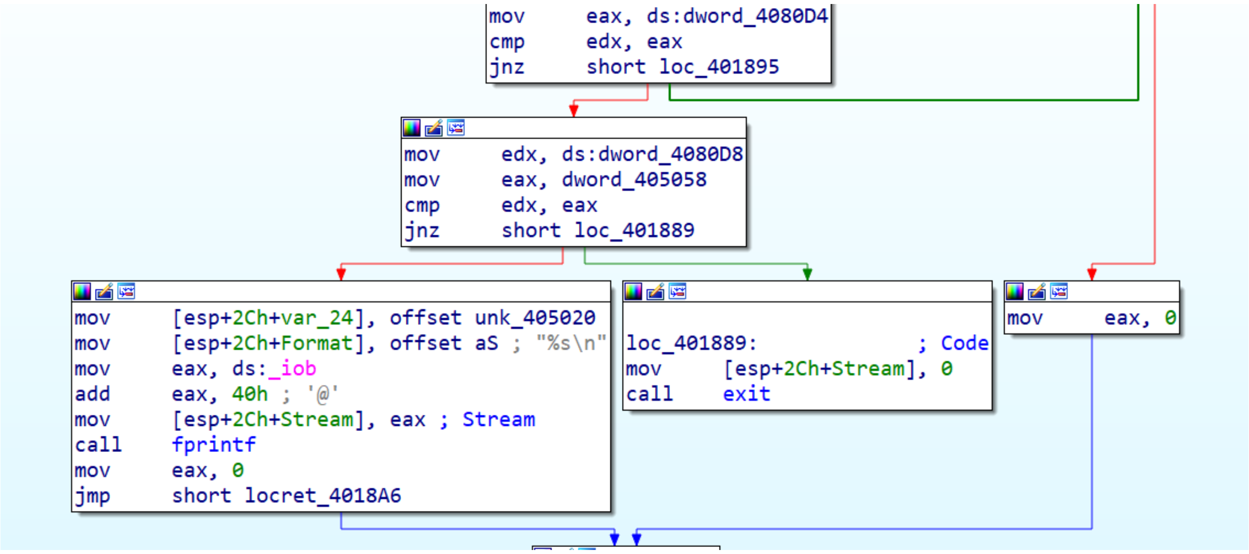
```
loc_401487:
mov     eax, [ebp+arg_0]
lea     edx, ds:0[eax*8]
mov     eax, [ebp+arg_4]
add     eax, edx
add     eax, offset unk_408020
movzx   eax, byte ptr [eax]
cmp     al, 58h ; 'X'
jnz     short loc_4014BA
```

```
mov     [esp+4+Buffer], offset aPathBlocked ; "path blocked!"
call    puts
mov     [esp+4+Buffer], 1 ; Code
call    exit
```

```
loc_4014BA:
nop
leave
retn
sub_40145D endp
```

Now, this looked promising, as we could see the memory previously initialized being accessed and while fooling with legal inputs before we repeatedly got these types of "error" messages. It was checking if some place in memory had X and if so printing "path blocked!" and if our arguments were bigger than 7 printing "out of bounds!".

We also found this



```
mov     eax, ds:dword_4080D4
cmp     edx, eax
jnz     short loc_401895
```

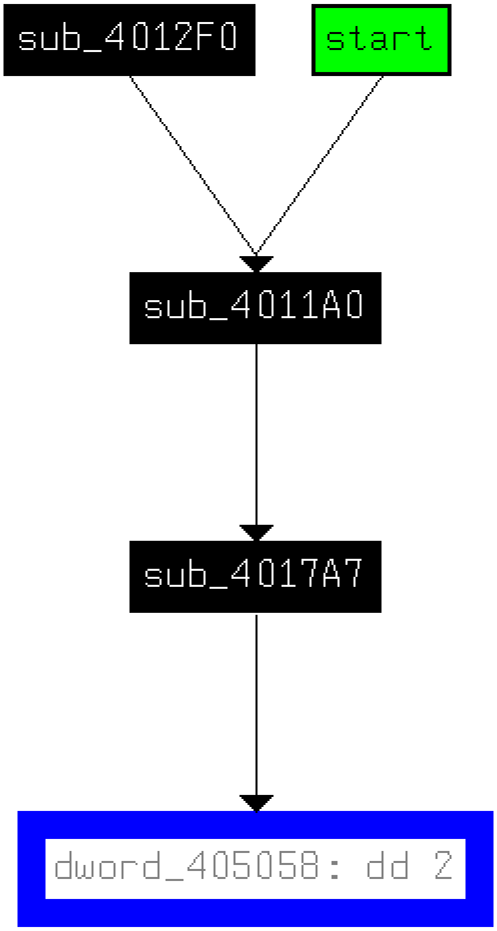
```
mov     edx, ds:dword_4080D8
mov     eax, dword_405058
cmp     edx, eax
jnz     short loc_401889
```

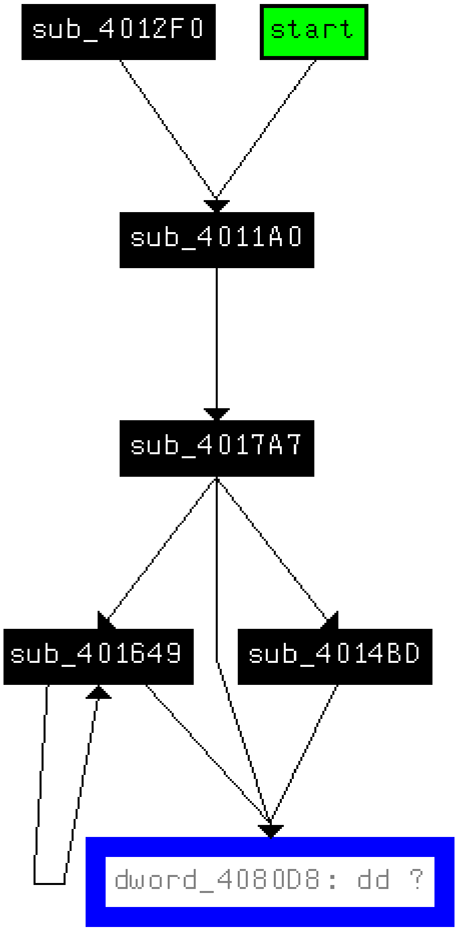
```
mov     [esp+2Ch+var_24], offset unk_405020
mov     [esp+2Ch+Format], offset aS ; "%s\n"
mov     eax, ds:_iob
add     eax, 40h ; '@'
mov     [esp+2Ch+Stream], eax ; Stream
call    fprintf
mov     eax, 0
jmp     short locret_4018A6
```

```
loc_401889:                 ; Code
mov     [esp+2Ch+Stream], 0
call    exit
```

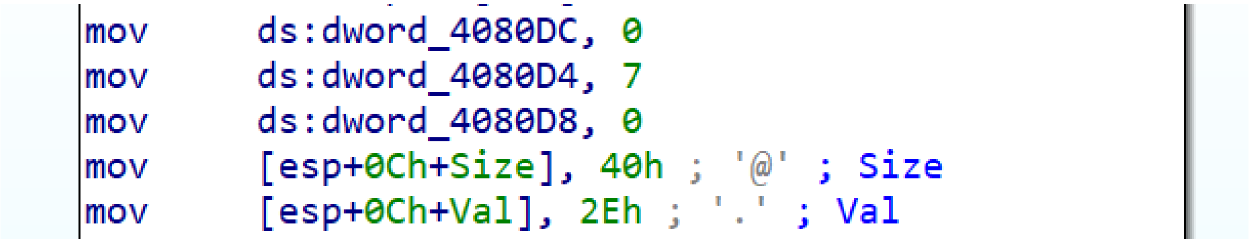
```
mov     eax, 0
```

Which looked as if-else clause to printing our desired output. As we can see, the clause compares two places in memory, and if they are equal, it prints. We asked ida for xrefs graphs to these places in memory and got the following:



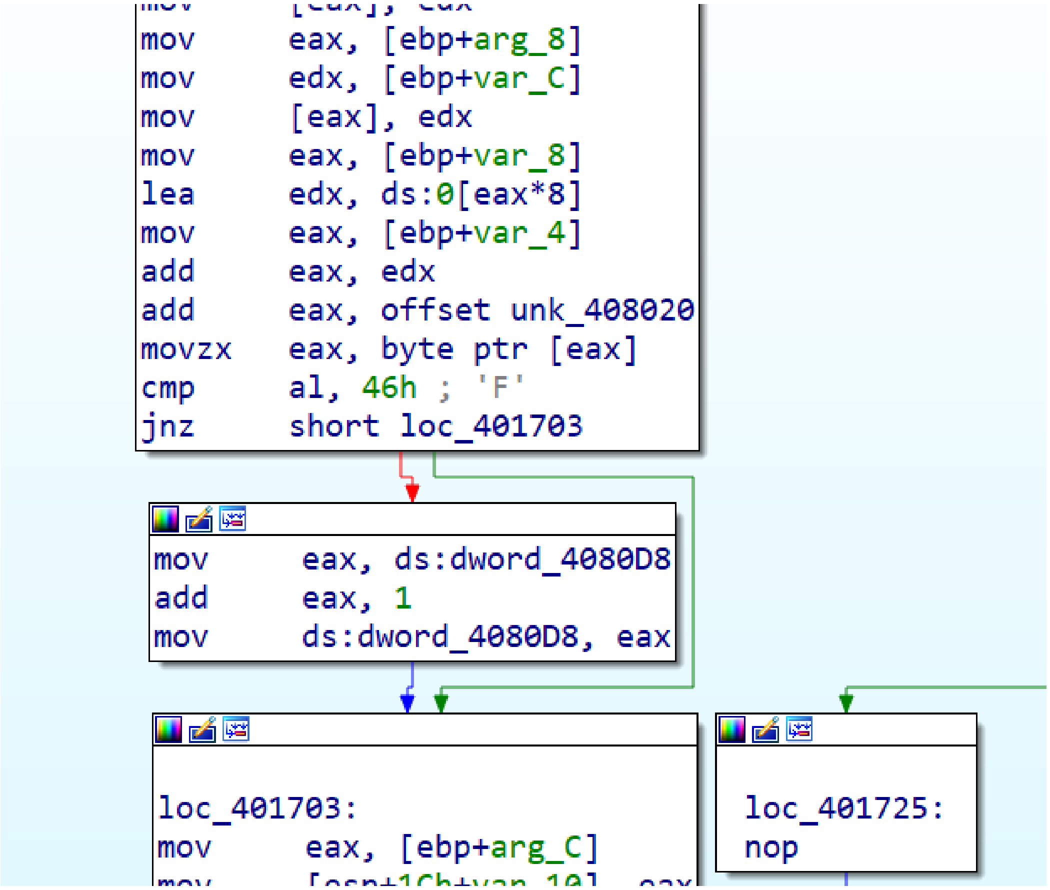


Which meant 405058 was always equal 2 and then we looked at the functions which changed 4080D8 and found



```
mov     ds:dword_4080DC, 0
mov     ds:dword_4080D4, 7
mov     ds:dword_4080D8, 0
mov     [esp+0Ch+Size], 40h ; '@' ; Size
mov     [esp+0Ch+Val], 2Eh ; '.' ; Val
```

One initializing it to zero



```
mov     eax, [ebp+arg_8]
mov     edx, [ebp+var_C]
mov     [eax], edx
mov     eax, [ebp+var_8]
lea     edx, ds:0[eax*8]
mov     eax, [ebp+var_4]
add     eax, edx
add     eax, offset unk_408020
movzx   eax, byte ptr [eax]
cmp     al, 46h ; 'F'
jnz     short loc_401703
```

```
mov     eax, ds:dword_4080D8
add     eax, 1
mov     ds:dword_4080D8, eax
```

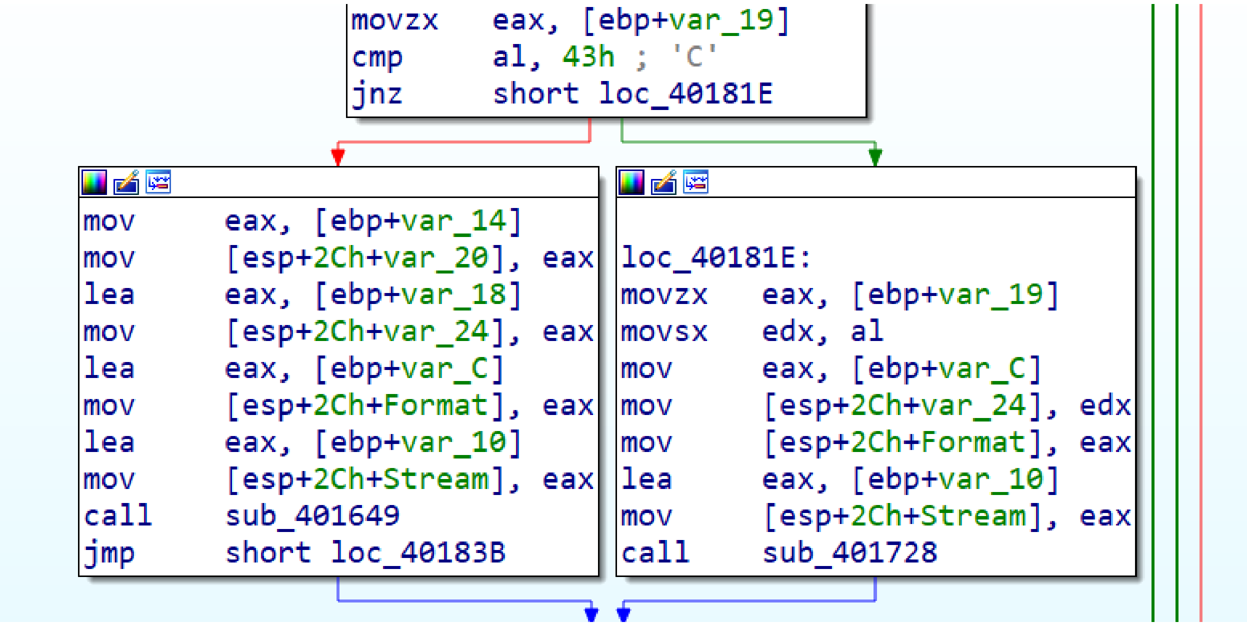
```
loc_401703:
mov     eax, [ebp+arg_C]
mov     [esp+1Ch+var_10], eax
```

```
loc_401725:
nop
```

And other adding 1 to if a certain place in memory was equal to F.

At the moment we had the understanding that there was a place in memory which eventually would be checked, and we needed it to find the value checked to be equal to F twice. Also, there were some legal inputs that looked to be interesting.

We dove in where our inputs were going, and we found



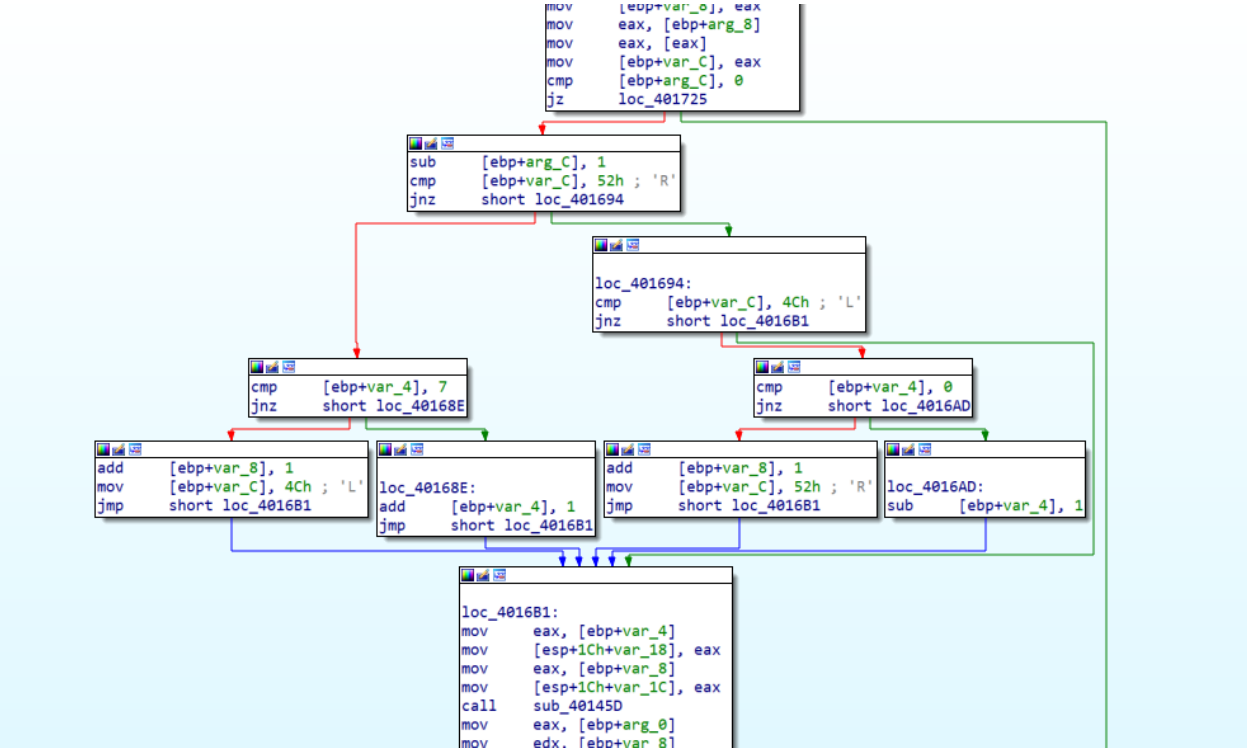
```
movzx   eax, [ebp+var_19]
cmp     al, 43h ; 'C'
jnz     short loc_40181E
```

```
mov     eax, [ebp+var_14]
mov     [esp+2Ch+var_20], eax
lea     eax, [ebp+var_18]
mov     [esp+2Ch+var_24], eax
lea     eax, [ebp+var_C]
mov     [esp+2Ch+Format], eax
lea     eax, [ebp+var_10]
mov     [esp+2Ch+Stream], eax
call    sub_401649
jmp     short loc_40183B
```

```

loc_40181E:
movzx   eax, [ebp+var_19]
movsx   edx, al
mov     eax, [ebp+var_C]
mov     [esp+2Ch+var_24], edx
mov     [esp+2Ch+Format], eax
lea     eax, [ebp+var_10]
mov     [esp+2Ch+Stream], eax
call    sub_401728
```

So, there was a check if we had ‘C’ or other things.

If we had ‘C’ we called this function



```
mov     [ebp+var_8], eax
mov     eax, [ebp+arg_8]
mov     eax, [eax]
mov     [ebp+var_C], eax
cmp     [ebp+arg_C], 0
jz      loc_401725
```

```
sub     [ebp+arg_C], 1
cmp     [ebp+var_C], 52h ; 'R'
jnz     short loc_401694
```

```

loc_401694:
cmp     [ebp+var_C], 4Ch ; 'L'
jnz     short loc_4016B1
```

```
cmp     [ebp+var_4], 7
jnz     short loc_40168E
```

```
cmp     [ebp+var_4], 0
jnz     short loc_4016AD
```

```
add     [ebp+var_8], 1
mov     [ebp+var_C], 4Ch ; 'L'
jmp     short loc_4016B1
```

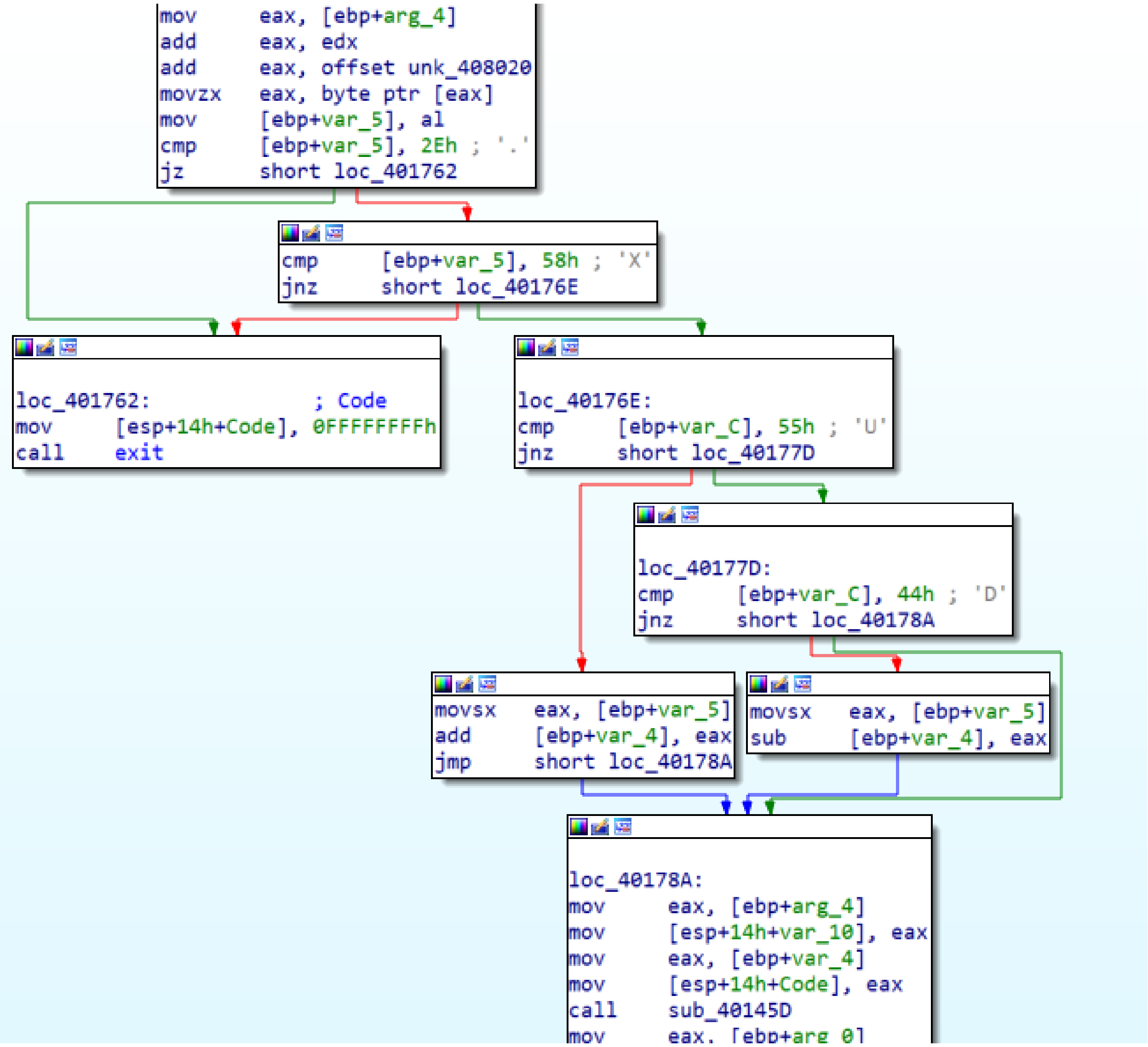
```

loc_40168E:
add     [ebp+var_4], 1
jmp     short loc_4016B1
```

```
add     [ebp+var_8], 1
mov     [ebp+var_C], 52h ; 'R'
jmp     short loc_4016B1
```

```

loc_4016AD:
sub     [ebp+var_4], 1
```

```

loc_4016B1:
mov     eax, [ebp+var_4]
mov     [esp+1Ch+var_18], eax
mov     eax, [ebp+var_8]
mov     [esp+1Ch+var_1C], eax
call    sub_40145D
mov     eax, [ebp+arg_0]
mov     edx, [ebp+var_8]
```

Where the last call called the function with the ‘error’ messages, so we figured we were on the right way. Now we saw this function was changing ‘L’ to ‘R’ and backwards if we achieved 7 and adding 1 to some value. And we thought it looked pretty suspicious that we had as of right now 4 “magic chars” which were R,L,U,D which sure sounded a lot like Right, Left, Up, Down.



```
mov     eax, [ebp+arg_4]
add     eax, edx
add     eax, offset unk_408020
movzx   eax, byte ptr [eax]
mov     [ebp+var_5], al
cmp     [ebp+var_5], 2Eh ; '.'
jz      short loc_401762
```

```
cmp     [ebp+var_5], 58h ; 'X'
jnz     short loc_40176E
```

```
loc_401762:                 ; Code
mov     [esp+14h+Code], 0FFFFFFFFh
call    exit
```

```
loc_40176E:
cmp     [ebp+var_C], 55h ; 'U'
jnz     short loc_40177D
```

```
loc_40177D:
cmp     [ebp+var_C], 44h ; 'D'
jnz     short loc_40178A
```

```
movsx   eax, [ebp+var_5]
add     [ebp+var_4], eax
jmp     short loc_40178A
```

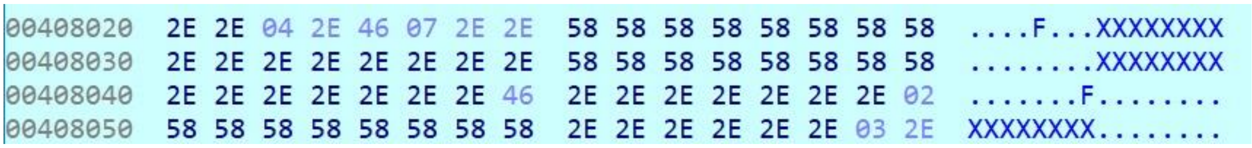
```
movsx   eax, [ebp+var_5]
sub     [ebp+var_4], eax
```

```
loc_40178A:
mov     eax, [ebp+arg_4]
mov     [esp+14h+var_10], eax
mov     eax, [ebp+var_4]
mov     [esp+14h+Code], eax
call    sub_40145D
mov     eax, [ebp+arg_0]
```

Once again, the last call called the function with the ‘error’ messages, so we were pretty sure that it represented a legal move check that checked if the place in memory had X. Now this function was accessing some place in memory, checking if it wasn’t X or a ‘.’

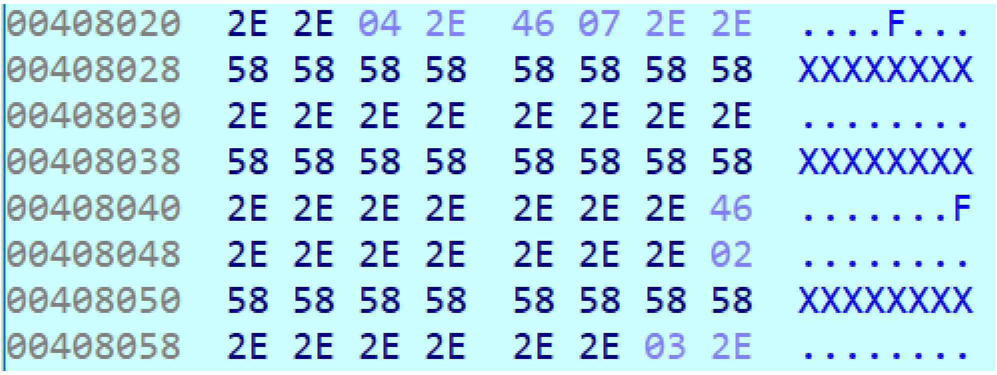
and if not adding/subtracting it from some value. This sounds a lot like Up and Down.

And finally, we decided we had to run the code and see how the memory looked and how the different input influenced the flow of the program. The memory looked like this



```
00408020  2E 2E 04 2E 46 07 2E 2E  58 58 58 58 58 58 58 58  ....F...XXXXXXXX
00408030  2E 2E 2E 2E 2E 2E 2E 2E  58 58 58 58 58 58 58 58  ........XXXXXXXX
00408040  2E 2E 2E 2E 2E 2E 2E 46  2E 2E 2E 2E 2E 2E 2E 02  .......F........
00408050  58 58 58 58 58 58 58 58  2E 2E 2E 2E 2E 2E 03 2E  XXXXXXXX........
```

And we finally figured that we had to look at it as 8x8 grid.



```
00408020  2E 2E 04 2E  46 07 2E 2E  ....F...
00408028  58 58 58 58  58 58 58 58  XXXXXXXX
00408030  2E 2E 2E 2E  2E 2E 2E 2E  ........
00408038  58 58 58 58  58 58 58 58  XXXXXXXX
00408040  2E 2E 2E 2E  2E 2E 2E 46  .......F
00408048  2E 2E 2E 2E  2E 2E 2E 02  ........
00408050  58 58 58 58  58 58 58 58  XXXXXXXX
00408058  2E 2E 2E 2E  2E 2E 03 2E  ........
```

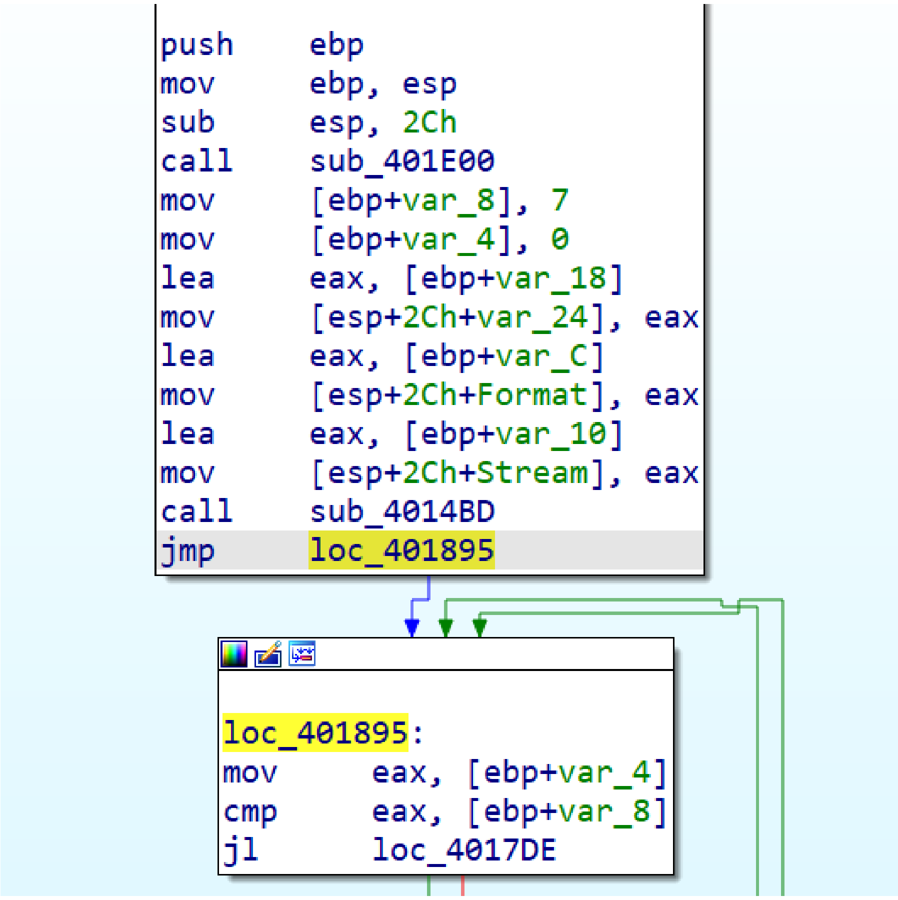
And we figured the laws to moving on that grid.

We found out that C# allowed us to move horizontally # steps. But if we went too far, we would go up a line and change direction. Initially our direction would be Right.

And D or U would make us go down or up the numbers of line we had in the current index.

Analyzing the step function of C# we realized as it worked by checking every step if legal and if it F we did not need to stop on F, only pass over it.

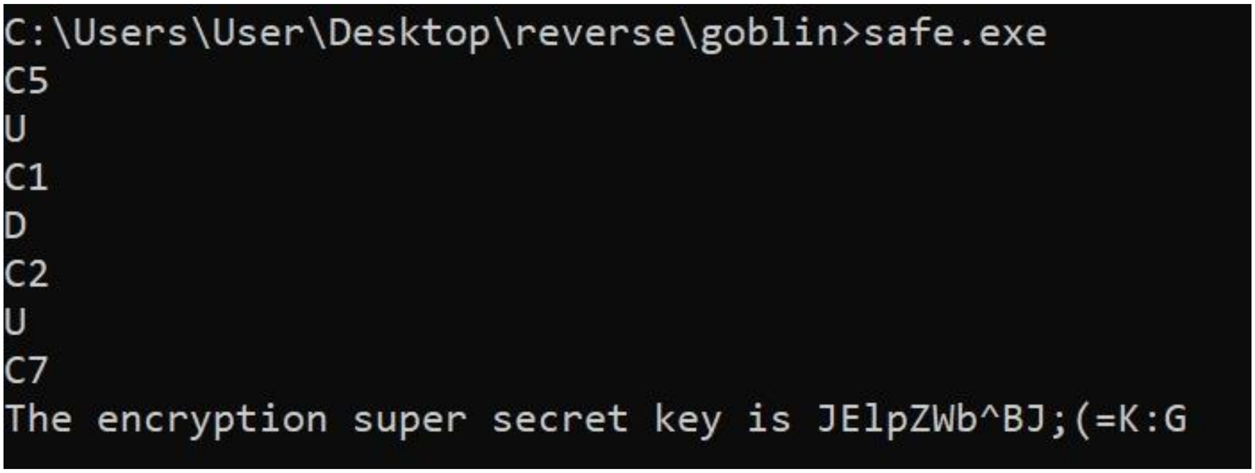
And we realized the main function limited our steps to at maximum 7 here



```
push    ebp
mov     ebp, esp
sub     esp, 2Ch
call    sub_401E00
mov     [ebp+var_8], 7
mov     [ebp+var_4], 0
lea     eax, [ebp+var_18]
mov     [esp+2Ch+var_24], eax
lea     eax, [ebp+var_C]
mov     [esp+2Ch+Format], eax
lea     eax, [ebp+var_10]
mov     [esp+2Ch+Stream], eax
call    sub_4014BD
jmp     loc_401895
```

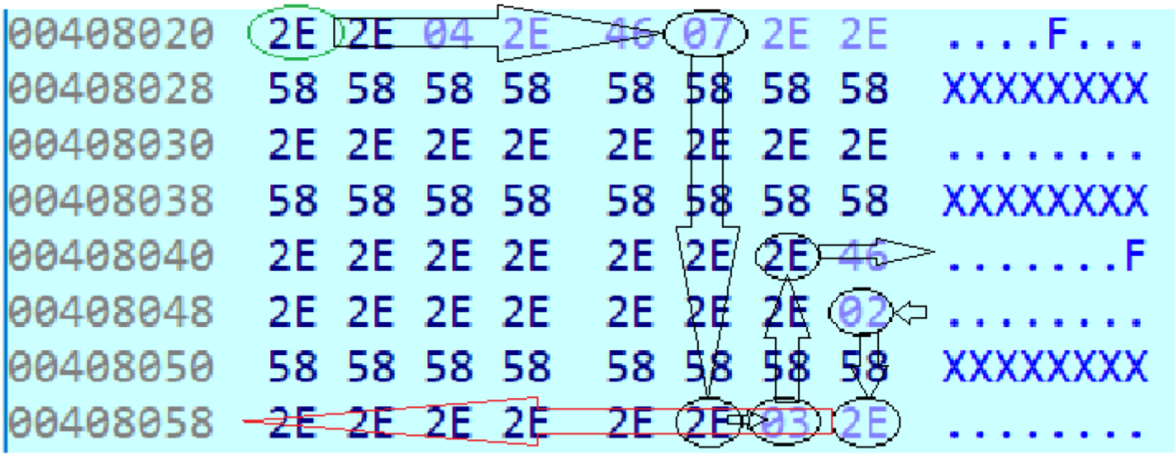
```
loc_401895:
mov     eax, [ebp+var_4]
cmp     eax, [ebp+var_8]
jl      loc_4017DE
```

Knowing, the laws of movement we traced a road to victory and found that the following input worked



```
C:\Users\User\Desktop\reverse\goblin>safe.exe
C5
U
C1
D
C2
U
C7
The encryption super secret key is JElpZWb^BJ;(=K:G
```

Let's briefly explain why it works



```
00408020  2E 2E 04 2E  46 07 2E 2E  ....F...
00408028  58 58 58 58  58 58 58 58  XXXXXXXX
00408030  2E 2E 2E 2E  2E 2E 2E 2E  ........
00408038  58 58 58 58  58 58 58 58  XXXXXXXX
00408040  2E 2E 2E 2E  2E 2E 2E 46  .......F
00408048  2E 2E 2E 2E  2E 2E 2E 02  ........
00408050  58 58 58 58  58 58 58 58  XXXXXXXX
00408058  2E 2E 2E 2E  2E 2E 03 2E  ........
```

Or in a table manner:

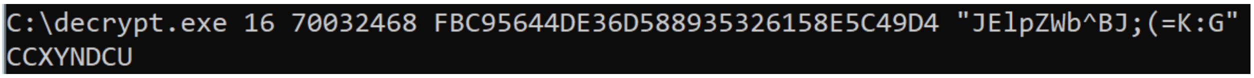
Current Address	Value in C.A.	Input	
408020	'.'	C5	Passed over F in 408024
408025	7	U	
40805D	'.'	C1	
40805E	3	D	
408046	'.'	C2	Passed over F in 408047
40804F	2	U	
40805F	'.'	C7	
408060			

Decrypt

After finishing with all 3 c(lash royale)atan characters, we executed the decrypt.exe file with the following arguments:
16 70032468

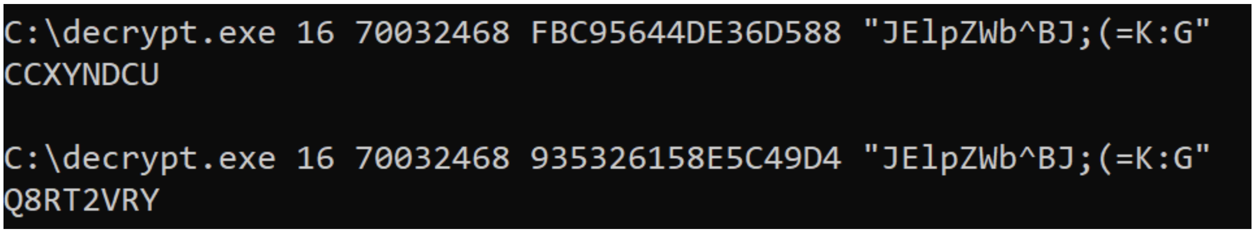
But it exited with the following error: “Error: Not enough arguments supplied.”
testing for how much arguments are required for dismissing an argument amount exception, we realized it was 4. Then we used the combinations of all of our other values we’ve found:
FBC95644DE36D588935326158E5C49D4
JElpZWb^BJ;(=K:G

And eventually received the following password:



```
C:\decrypt.exe 16 70032468 FBC95644DE36D588935326158E5C49D4 "JElpZWb^BJ;(=K:G"
CCXYNDCU
```

After a failed attempt to open the shared safe, and a strange mistake of removing a few of the last characters in the password, we realized that we still had job to do. Investigating the decrypt.exe file with IDA and trying to understand which characters affect the result, we noticed that the file considered only half of the hashed password, which can lead to the option that we need to decrypt the second half on a different execution:



```
C:\decrypt.exe 16 70032468 FBC95644DE36D588 "JElpZWb^BJ;(=K:G"
CCXYNDCU

C:\decrypt.exe 16 70032468 935326158E5C49D4 "JElpZWb^BJ;(=K:G"
Q8RT2VRY
```

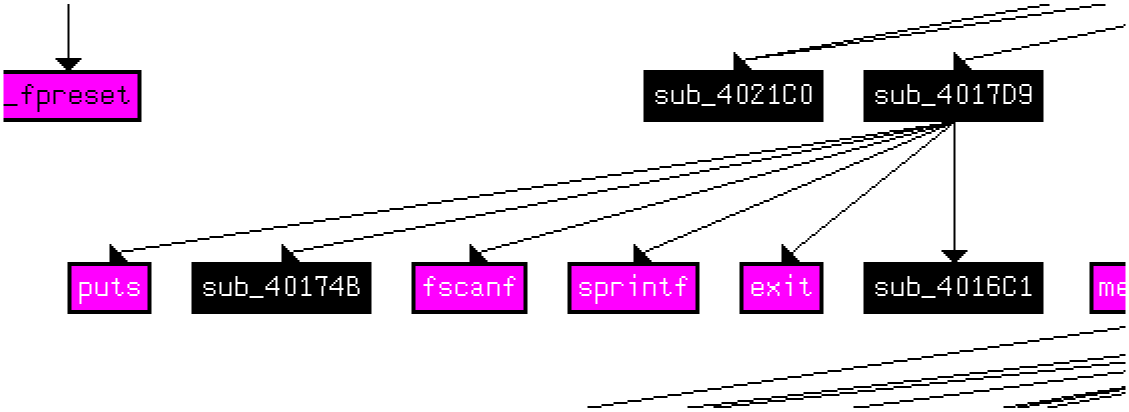
We then tried the new password (the concatenate of both outputs) which is CCXYNDCUQ8RT2VRY and we successfully managed to open the shared safe!

Sheep

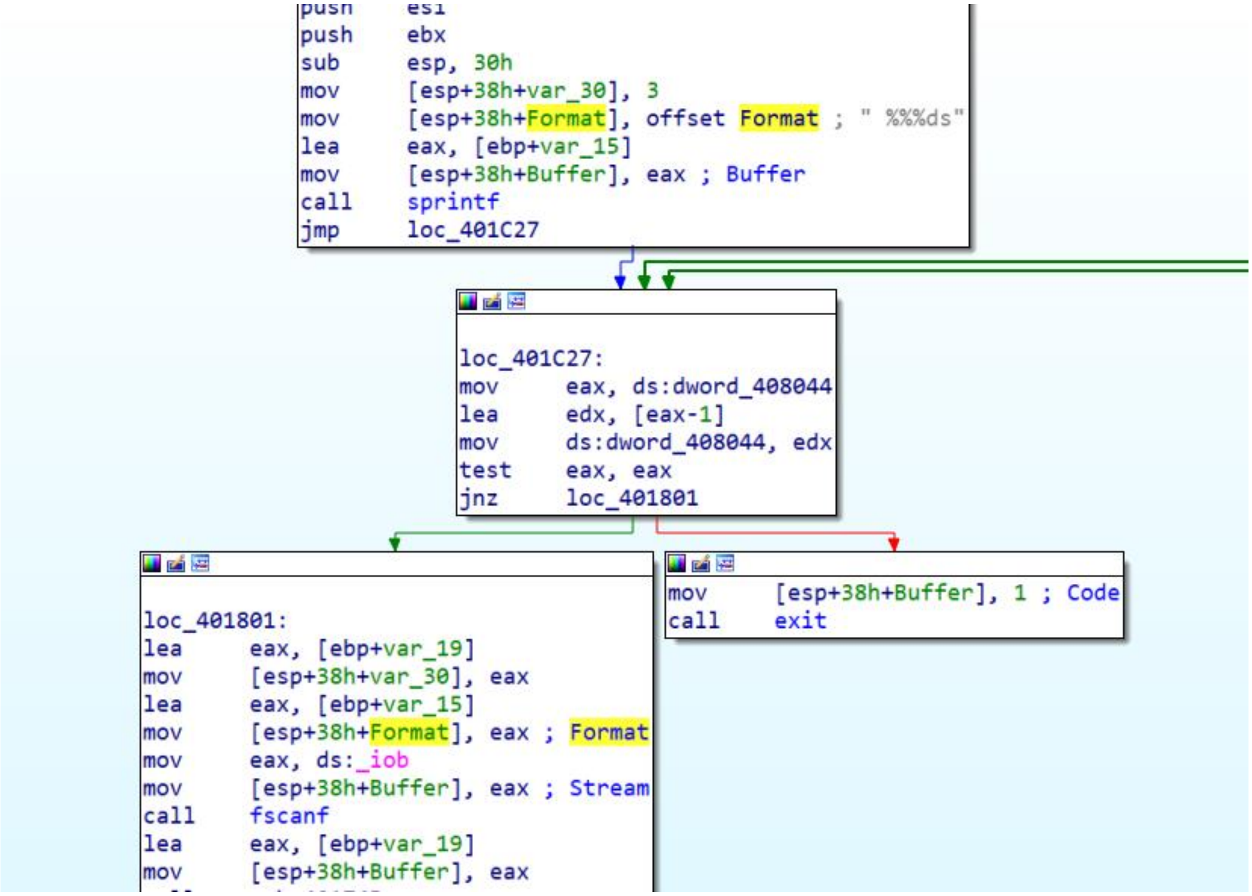
We just need to solve riddles of 3 characters and finish this exercise
We just need to decrypt the password to open the shared safe and finish this exercise
We just need to open the sheep file to see the answer and finish this exercise
imgflip.com

The moment we unzipped the file and found a picture we knew we needed to use the analyzer.exe. It outputted a file sheep.out, and it didn't look like text, so we tried opening on IDA and it was an executable.

We straight up made a graph of function calls and found that there is only function that really does input/output



So there we went



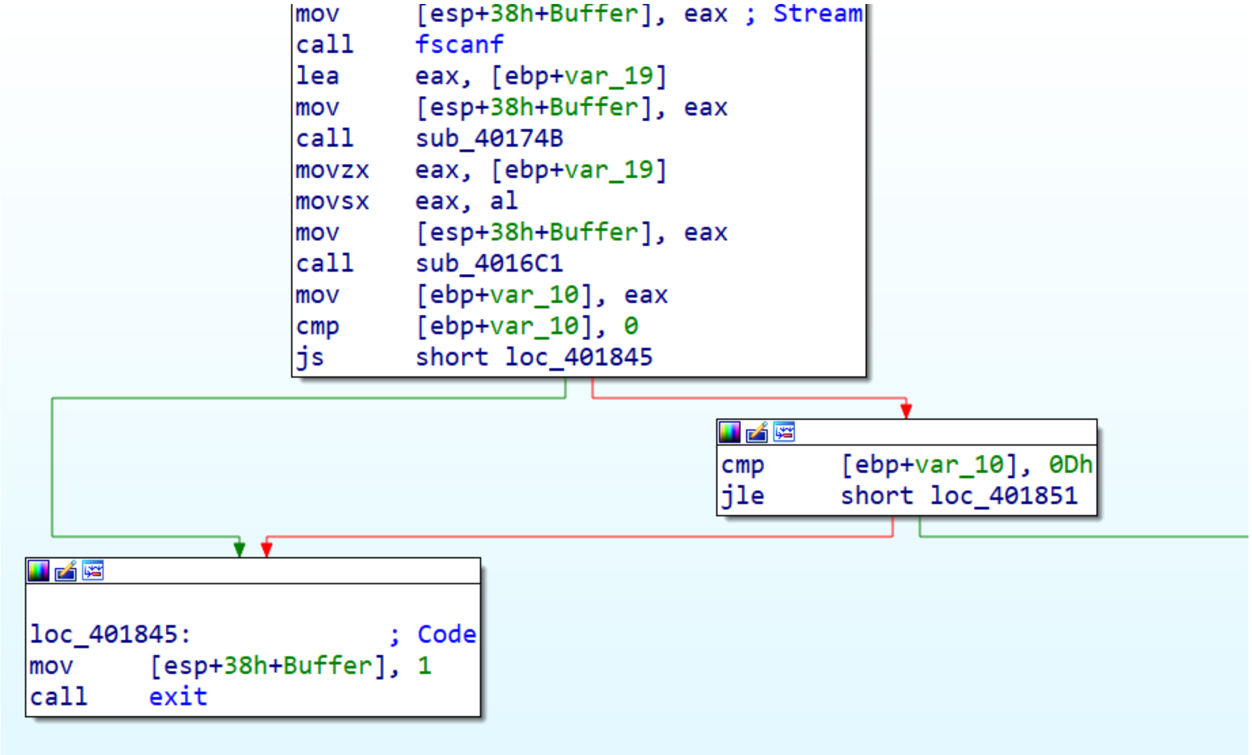
```
push    esi
push    ebx
sub     esp, 30h
mov     [esp+38h+var_30], 3
mov     [esp+38h+Format], offset Format ; " %%%ds"
lea     eax, [ebp+var_15]
mov     [esp+38h+Buffer], eax ; Buffer
call    sprintf
jmp     loc_401C27
```

```
loc_401C27:
mov     eax, ds:dword_408044
lea     edx, [eax-1]
mov     ds:dword_408044, edx
test    eax, eax
jnz     loc_401801
```

```
loc_401801:
lea     eax, [ebp+var_19]
mov     [esp+38h+var_30], eax
lea     eax, [ebp+var_15]
mov     [esp+38h+Format], eax ; Format
mov     eax, ds:_iob
mov     [esp+38h+Buffer], eax ; Stream
call    fscanf
lea     eax, [ebp+var_19]
mov     [esp+38h+Buffer], eax
```

```
mov     [esp+38h+Buffer], 1 ; Code
call    exit
```

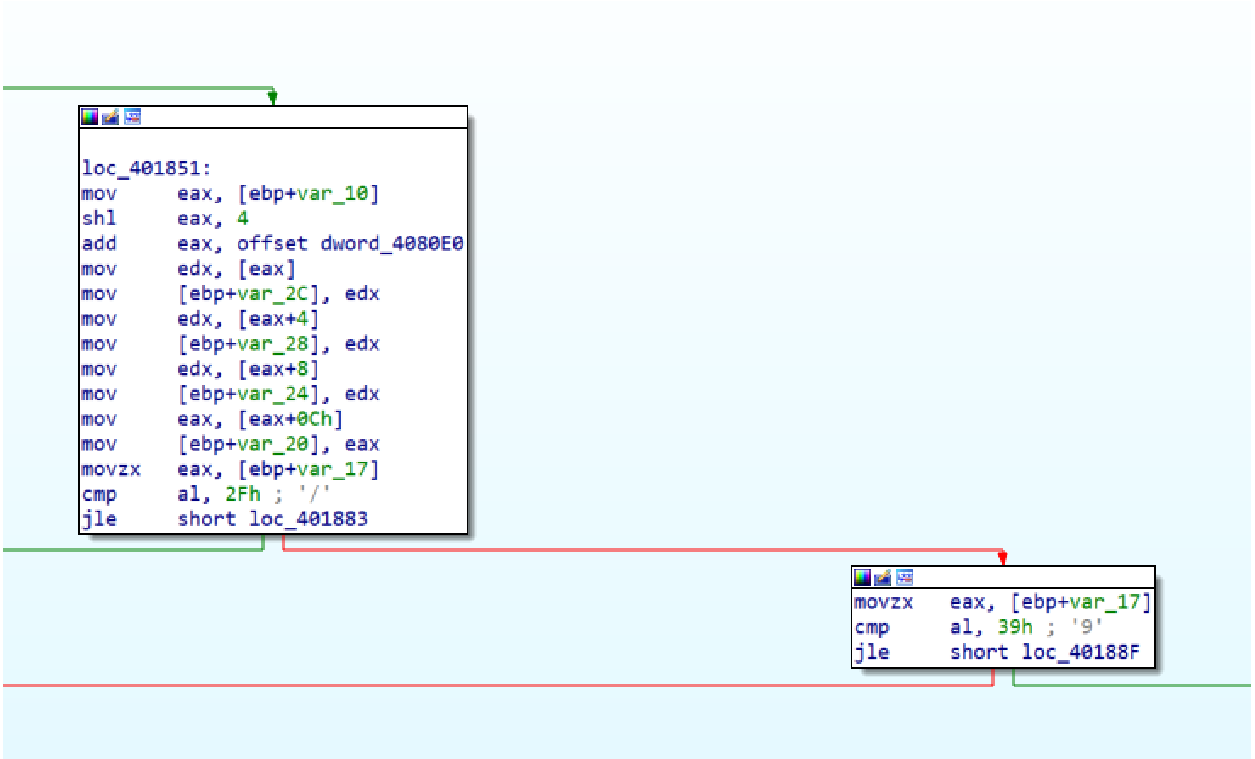
Looking at the function we found out that the input are 3-character strings each time



```
mov     [esp+38h+Buffer], eax ; Stream
call    fscanf
lea     eax, [ebp+var_19]
mov     [esp+38h+Buffer], eax
call    sub_40174B
movzx   eax, [ebp+var_19]
movsx   eax, al
mov     [esp+38h+Buffer], eax
call    sub_4016C1
mov     [ebp+var_10], eax
cmp     [ebp+var_10], 0
js      short loc_401845
```

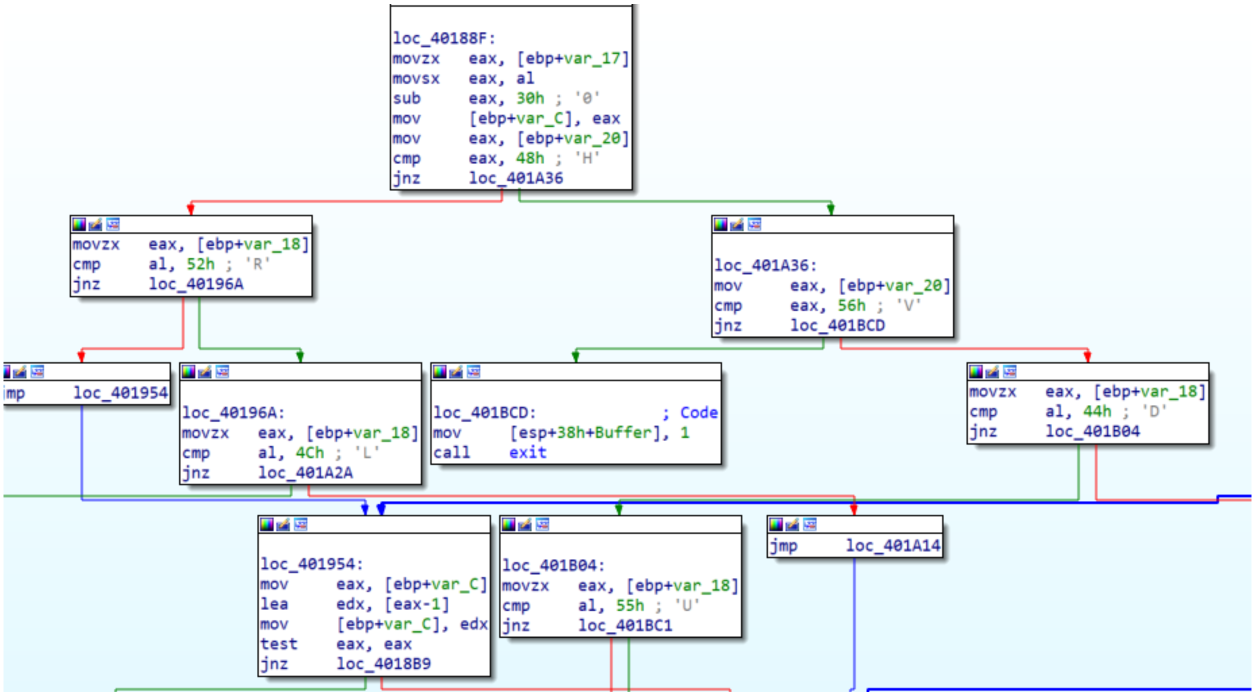
```
cmp     [ebp+var_10], 0Dh
jle     short loc_401851
```

```
loc_401845:                     ; Code
mov     [esp+38h+Buffer], 1
call    exit
```



```
loc_401851:
mov     eax, [ebp+var_10]
shl     eax, 4
add     eax, offset dword_4080E0
mov     edx, [eax]
mov     [ebp+var_2C], edx
mov     edx, [eax+4]
mov     [ebp+var_28], edx
mov     edx, [eax+8]
mov     [ebp+var_24], edx
mov     eax, [eax+0Ch]
mov     [ebp+var_20], eax
movzx   eax, [ebp+var_17]
cmp     al, 2Fh ; '/'
jle     short loc_401883
```

```
movzx   eax, [ebp+var_17]
cmp     al, 39h ; '9'
jle     short loc_40188F
```



```
loc_40188F:
movzx   eax, [ebp+var_17]
movsx   eax, al
sub     eax, 30h ; '0'
mov     [ebp+var_C], eax
mov     eax, [ebp+var_20]
cmp     eax, 48h ; 'H'
jnz     loc_401A36
```

```
movzx   eax, [ebp+var_18]
cmp     al, 52h ; 'R'
jnz     loc_40196A
```

```
loc_401A36:
mov     eax, [ebp+var_20]
cmp     eax, 56h ; 'V'
jnz     loc_401BCD
```

```
jmp     loc_401954
```

```
loc_40196A:
movzx   eax, [ebp+var_18]
cmp     al, 4Ch ; 'L'
jnz     loc_401A2A
```

```
loc_401BCD:                 ; Code
mov     [esp+38h+Buffer], 1
call    exit
```

```
movzx   eax, [ebp+var_18]
cmp     al, 44h ; 'D'
jnz     loc_401B04
```

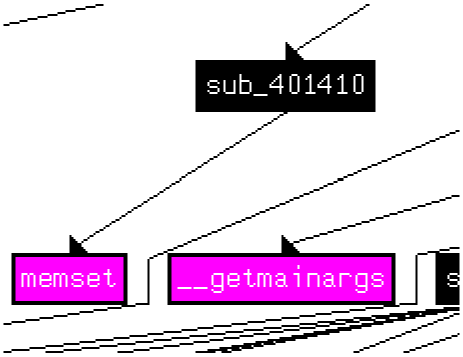
```
loc_401954:
mov     eax, [ebp+var_C]
lea     edx, [eax-1]
mov     [ebp+var_C], edx
test    eax, eax
jnz     loc_4018B9
```

```
loc_401B04:
movzx   eax, [ebp+var_18]
cmp     al, 55h ; 'U'
jnz     loc_401BC1
```

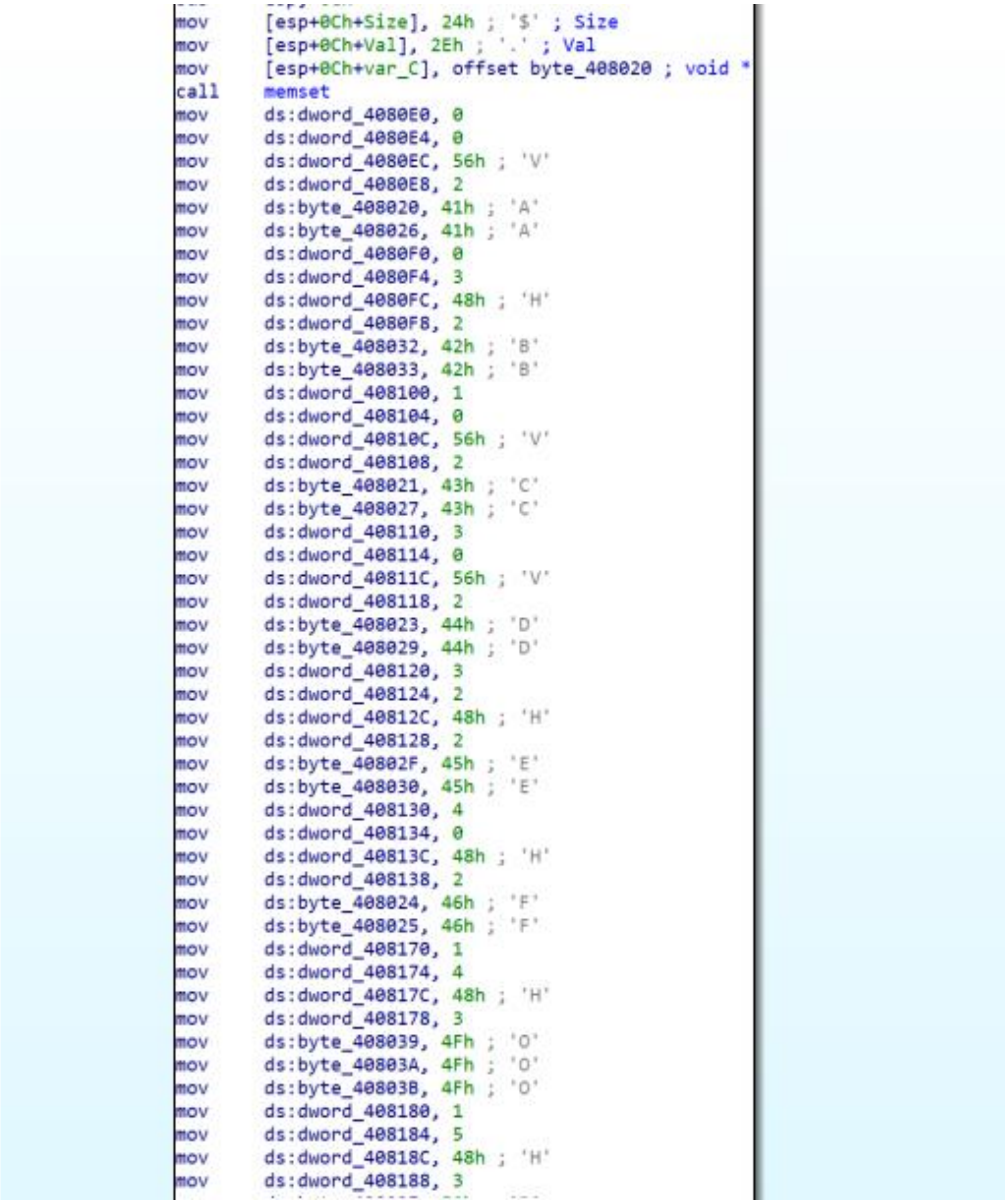
```
jmp     loc_401A14
```

We figured the last character had to be a digit, the middle one needed to be one of R, L, D, U, and the first one had a jump table that at first we could not make sense of it. The R, L, D, U instantly throwed us back to the goblin exercise and we started looking for something to call a grid.

As last time the memset function was used, we looked to see maybe the grid was set again with memset.



Looking there we found the following



```
mov     [esp+0Ch+Size], 24h ; '$' ; Size
mov     [esp+0Ch+Val], 2Eh ; '.' ; Val
mov     [esp+0Ch+var_C], offset byte_408020 ; void *
call    memset
mov     ds:dword_4080E0, 0
mov     ds:dword_4080E4, 0
mov     ds:dword_4080EC, 56h ; 'V'
mov     ds:dword_4080E8, 2
mov     ds:byte_408020, 41h ; 'A'
mov     ds:byte_408026, 41h ; 'A'
mov     ds:dword_4080F0, 0
mov     ds:dword_4080F4, 3
mov     ds:dword_4080FC, 48h ; 'H'
mov     ds:dword_4080F8, 2
mov     ds:byte_408032, 42h ; 'B'
mov     ds:byte_408033, 42h ; 'B'
mov     ds:dword_408100, 1
mov     ds:dword_408104, 0
mov     ds:dword_40810C, 56h ; 'V'
mov     ds:dword_408108, 2
mov     ds:byte_408021, 43h ; 'C'
mov     ds:byte_408027, 43h ; 'C'
mov     ds:dword_408110, 3
mov     ds:dword_408114, 0
mov     ds:dword_40811C, 56h ; 'V'
mov     ds:dword_408118, 2
mov     ds:byte_408023, 44h ; 'D'
mov     ds:byte_408029, 44h ; 'D'
mov     ds:dword_408120, 3
mov     ds:dword_408124, 2
mov     ds:dword_40812C, 48h ; 'H'
mov     ds:dword_408128, 2
mov     ds:byte_40802F, 45h ; 'E'
mov     ds:byte_408030, 45h ; 'E'
mov     ds:dword_408130, 4
mov     ds:dword_408134, 0
mov     ds:dword_40813C, 48h ; 'H'
mov     ds:dword_408138, 2
mov     ds:byte_408024, 46h ; 'F'
mov     ds:byte_408025, 46h ; 'F'
mov     ds:dword_408170, 1
mov     ds:dword_408174, 4
mov     ds:dword_40817C, 48h ; 'H'
mov     ds:dword_408178, 3
mov     ds:byte_408039, 4Fh ; 'O'
mov     ds:byte_40803A, 4Fh ; 'O'
mov     ds:byte_40803B, 4Fh ; 'O'
mov     ds:dword_408180, 1
mov     ds:dword_408184, 5
mov     ds:dword_40818C, 48h ; 'H'
mov     ds:dword_408188, 3
```

We saw that memset laid down 36 bytes, which could be seen as 6x6 square.

408020	A	C	.	D	F	F
408026	A	C	X	D	.	.
40802C	.	.	X	E	E	.
408032	B	B	Q	Q	Q	R
408038	.	O	O	O	.	R
40803E	.	P	P	P	.	R

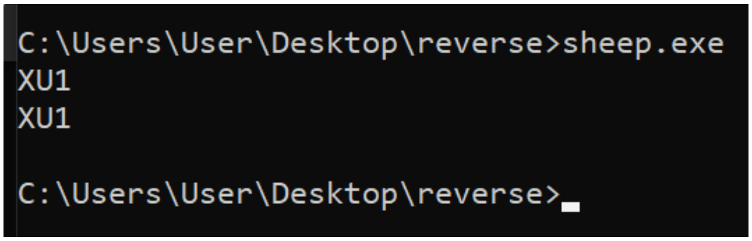
We realized that always the same letter would be grouped together and then as we thought more about it, we started seeing a deeper pattern.

A	C		D	F	F
A	C	X	D		
		X	E	E	
B	B	Q	Q	Q	R
	O	O	O		R
	P	P	P		R

Now, this looks a lot like rush hour, but is it? We had to test. We tried some inputs, like AD1, RU2, that were supposed to work, and they indeed did, and some that were not and they did not work.

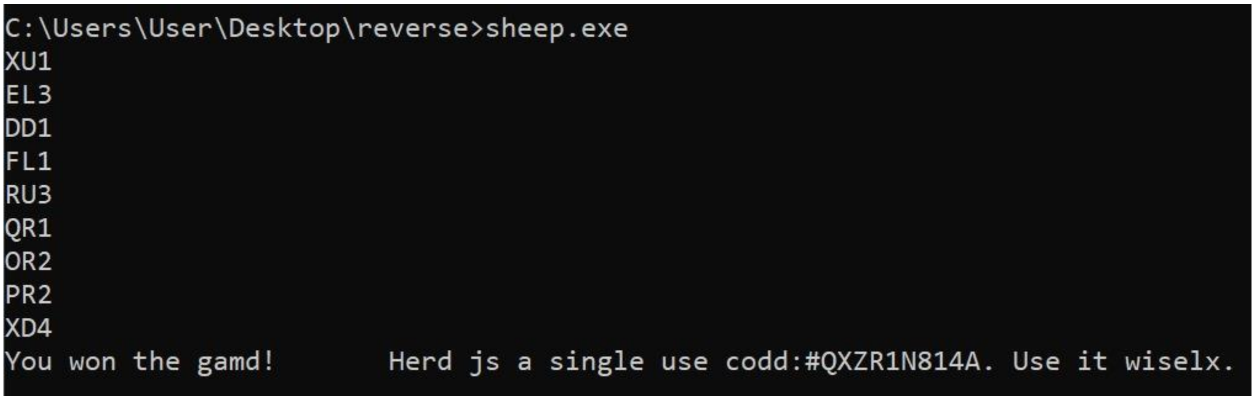
We tried harder, we figured out from standard rush hour that either X or E needed to be the objective so we started trying to make it exit.

We started with X. As we did not know orientation, we tried the simple XU1, it waited for more input so we thought maybe it needs to cross, but this caused the program to stop running:



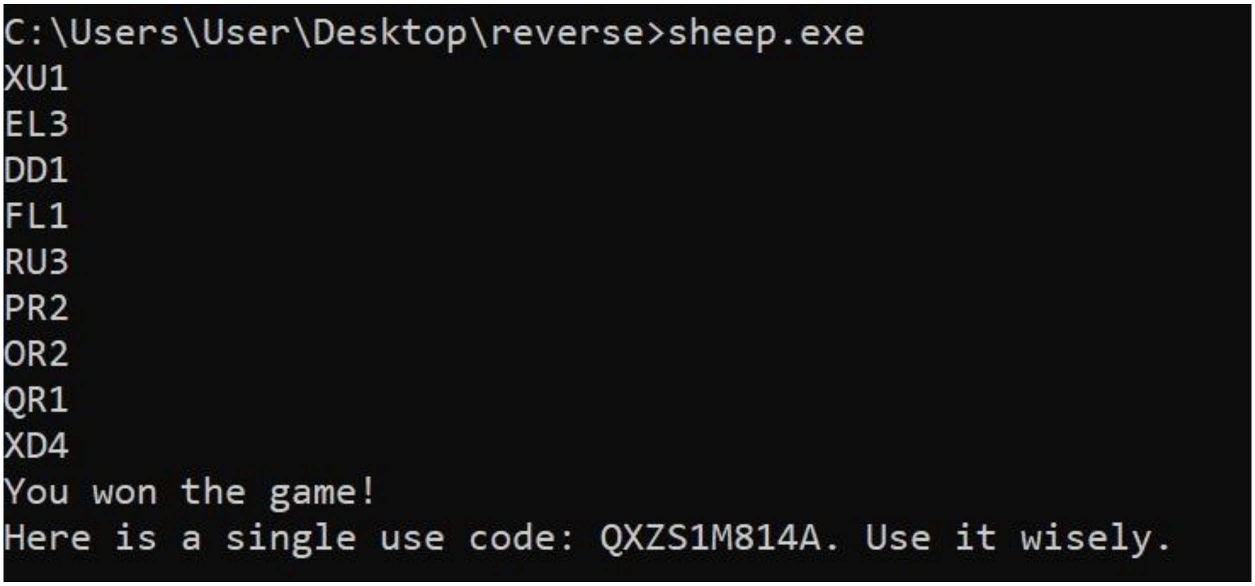
```
C:\Users\User\Desktop\reverse>sheep.exe
XU1
XU1

C:\Users\User\Desktop\reverse>_
```

So, we realized that it needed to go down. We created a smart solution, but we got a weird result:



```
C:\Users\User\Desktop\reverse>sheep.exe
XU1
EL3
DD1
FL1
RU3
QR1
OR2
PR2
XD4
You won the gamd!       Herd js a single use codd:#QXZR1N814A. Use it wiselx.
```

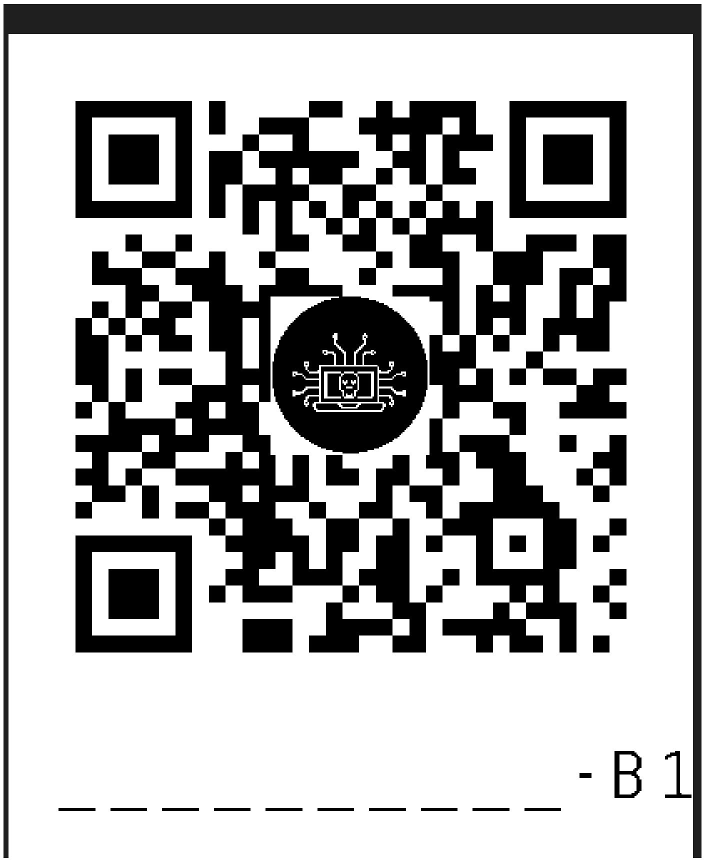
This looked like the answer so we knew we were in the right track, but for some reason it did not work. We started trying similar solutions, until we got



```
C:\Users\User\Desktop\reverse>sheep.exe
XU1
EL3
DD1
FL1
RU3
PR2
OR2
QR1
XD4
You won the game!
Here is a single use code: QXZS1M814A. Use it wisely.
```

Now, we had the code.

As the codes.pdf talked about a postfix, and we had one from using analyzer.exe on qrcode.png



__________-B1

We used the code from sheep.exe and this postfix and

We finished, and sheep was restored to the board

IGNORING THAT IN THE NEXT COUPLE
DAYS WE'LL HAVE ANOTHER ASSIGNMENT
imgflip.com