

Passwords obtained from the last assignment:

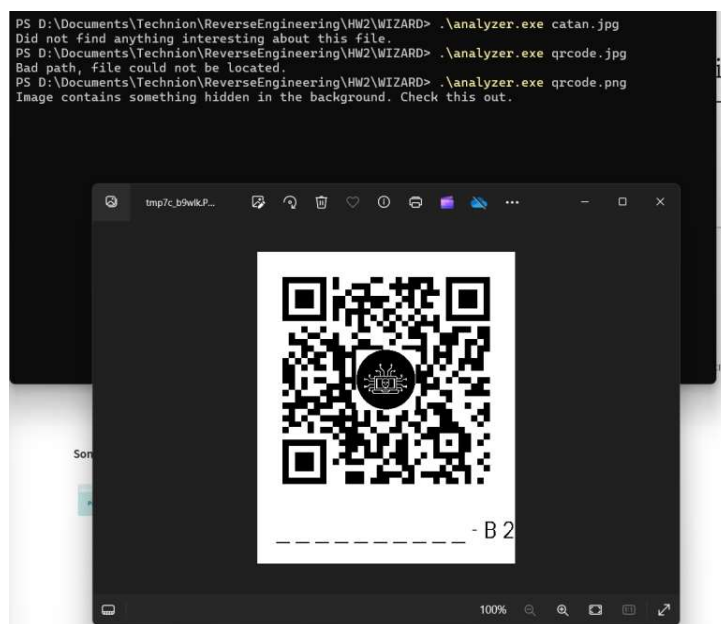
User	Password	Games lost	Games won
Wizard	LLOY7NWXVMR9KNLH	549/560	11
Goblin	ZFBNOC39V6UTPMNG	569/574	5
GIANT	52420OSKO1BBMLEH	444/458	14
ARCHER	62OA14UGACMN9OA6	638/645	7

Wizard's safe:

We downloaded the zip files.

Immediately ran the analyzer on catan.jpg but found nothing, but when running the analyzer on qrcode.png the last 2 digits of maybe a code revealed to us "-B2".

This will help us in the final stage when we enter the code to reveal the board.



After exploring the images we used IDA to disassemble the executable safe.exe and after reading it and playing with the input we found:



So we used python to solved the system of equation above:

```
from itertools import permutations
1 usage
def find_solution():
    for perm in perms:
        EC, E8, E4, E0, DC, D8, D4, D0, CC, C8, C4, C0 = perm
        if (
            E8 + E4 + E0 + DC == 26 and
            D0 + CC + C8 + C4 == 26 and
            EC + E4 + D8 + D0 == 26 and
            EC + E0 + D4 + C4 == 26 and
            E8 + D8 + CC + C0 == 26 and
            DC + D4 + C8 + C0 == 26 and
            EC + E8 + DC + D0 + C4 + C0 == 26
        ):
            solution = perm
            break
    print(solution)

if __name__ == '__main__':
    variables = ['EC', 'E8', 'E4', 'E0', 'DC', 'D8', 'D4', 'D0', 'CC', 'C8', 'C4', 'C0']
    values = range(1, 13)
    perms = permutations(values)
    find_solution()
```

Solution:

```
(1, 2, 10, 9, 5, 7, 12, 8, 11, 3, 4, 6)
```

Using this:

We will match each variable to the right num:

For EC to give me '1' input should be : '8'

For E8 to give me '1' input should be : '5'

For E4 to give me '1' input should be : '1'

For E0 to give me '1' input should be : '11'

For DC to give me '1' input should be : '10'

For D8 to give me '1' input should be : '2'

For D4 to give me '1' input should be : '3'

For D0 to give me '1' input should be : '7'

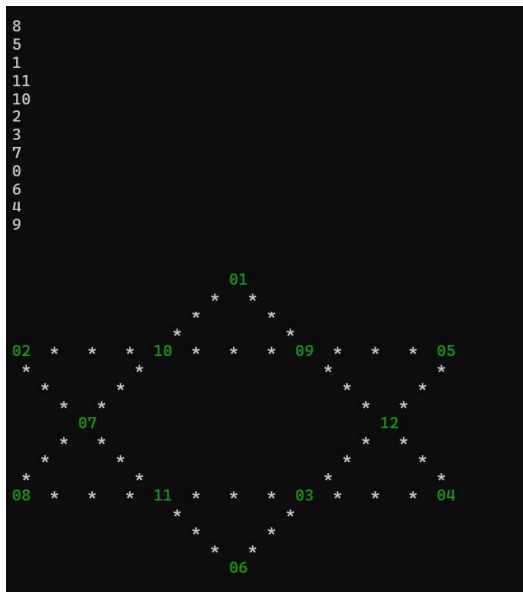
For CC to give me '1' input should be : '0'

For C8 to give me '1' input should be : '6'

For C4 to give me '1' input should be : '4'

For C0 to give me '1' input should be : '9'

First input is : 8 5 1 11 10 2 3 7 0 6 4 9



The hashed password is: cc5e|

Now we move on to new function:

We need to insert a string which after the next code will be equal to

**GMEKKODGBIHBNDLNNNGFANMGHBB**

so we wrote another program to help us decrypt that string this time in C:

```

1  #include <stdio.h>
2  #include <stdint.h>
3  #include <string.h>
4
5  char* encrypt(char* buffer, int size) {
6      for (int i = 0; i < size; ++i) {
7          buffer[i] = buffer[i] - 0x41;
8      }
9
10     for (int i = 0; i < size; ++i) {
11         buffer[i] = buffer[i] ^ buffer[(i + 1) % size];
12     }
13
14     for (int i = 0; i < size; ++i) {
15         buffer[i] = buffer[i] + 0x41;
16     }
17     return buffer;
18 }
19
20 int main() {
21     // DIEAEOEKJPOGBANGFODODFAANBHAH
22     char old[30] = {'A'};
23     char new[30] = "GMEKKODGBIHBNDLNNNGFANMGHBB";
24
25     for (int i = 0; i < 26; ++i) {
26         old[0] = 'A' + i;
27         for (int idx = 1; idx < 29; ++idx) {
28             old[idx] = ((old[idx - 1] - 0x41) ^ (new[idx - 1] - 0x41)) + 0x41;
29         }
30         if (encrypt(old, 29)[28] == 'B') {
31             printf("%s\n", old);
32             break;
33         }
34         // printf("%s\n", encrypt(old, 29));
35     }
36 }

```

The decrypted string is: OIEAEOEKJPOGBANGFODODFAANBHAH

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

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

The hashed password is: cc5e
OIEAEOEKJPOGBANGFODODFAANBHAH
PS D:\Documents\Technion\ReverseEngineering\HW2\WIZARD> |
```

However, after we insert the program is exited.

Then we looked we see that there is a compare between the return address to another address located on the stack - at the end of our second input.

That means we need to cause a buffer override to add another 16 bytes to make the compare happen in the right area of the stack - where the return address is.

Before:

0060FEF4	47000009	...G
0060FEF8	4B45454D	MEEK
0060FEFC	47444F4B	KODG
0060FF00	42484942	BIHB
0060FF04	4C444C4E	NLDL
0060FF08	474E4E4E	NNNG
0060FF0C	4D4E4146	FANM
0060FF10	42484847	GHHB
0060FF14	00000000	....
0060FF18	00000004	....
0060FF1C	003E6000	.`>.
0060FF20	0060FF28	(.`.
0060FF24	00401485	..@.

OIEAEOEKJPOGBANGFODODFAANBHAHAAAAAAAAAAAAAAAAAAAA

OIEAEOEKJPOGBANGFODODFAANBHAHAAAAAAAAAAAAAAAAAAAA



The hashed password is: cc5ed245C68AB4140f0214cbF51A16BC

Giant:

We converted the code from assembly to C:

```
#include <stdio.h>
#include <stdlib.h>

int main() {
    int Format = 0xdeadbeef;
    int var_24 = 0;
    int var_20 = 0;
    int var_1C = 0;
    int var3 = 0;
    int var1 = 0;
    int var2 = 0;
    int AddressX = 0;
    char var_5 = 0;
    int var_4 = 0;

    scanf("%x", &AddressX);

    if (AddressX < 0) {
        AddressX = -AddressX;
    }

    scanf("%d %d %d", &var1, &var2, &var3);

    var_4 = (var1 * var1 * -0x1b) + (var1 * 0xffffef20) - 0x2a2ff;
    var_5 = (char)var3;

    if (var3 < 0 || var_5 == 0xd1) {
        exit(1);
    }

    if (var_4 == 1 && var2 < 0 && var2 - var_4 <= 0 && AddressX < 0) {
        printf("%d\n", 9);
        printf("%d\n", (char)var1);
    } else {
        exit(1);
    }

    return 0;
}
```

**equation:**  $[(X^2)*(-27)] + X(-4320) - 172799 = 1$

$$-27x^2 - 4320x - 172799 = 1$$

פתרון

$$x = -80$$

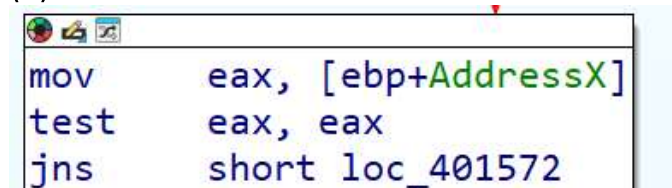
(1) = -80

(2) = 2147483648 = 80000000h

This is the lowest number that variable of 4 bytes can represent.

80000000h - 1h = 7FFFFFFFh which is a positive number.

(3) D1h = 209



```
mov     eax, [ebp+AddressX]
test    eax, eax
jns     short loc_401572
```

Now we need HEXdecimal number that is negative which after multiple by (-1) will stay negative , because we want that after 'test eax,eax' we will get signed number:

80000000h

will do the job.

```
80000000
209 2147483648 -80
The encryption para|
```

But now we had a problem - IDA after executing the handler of the exception exited the program,  
meaning we can't debug the 3rd function.

#### Input until now:

80000000

209 2147483648 -80 NULL

Function 3:

we need to insert 5 bytes HEX number (02%h) .

This input will be memcopy to address 401AAE which are filled with nops at the moment.

This address is located in the code section - which means we will modify the code and its possible not only because of memcopy but because of VirtualProtect function as we understand.

Using : "AA AA AA AA AA" as input created a exception and nothing came out from it.

We did understand that this is a recursive function.

Then we saw that one of the arguments is the address of the 8 numbers which added to the stack.

Meaning its an array of 8 numbers and the recursive function operates MERGESORT on this array.

Before the 5 NOPs we have 4 pushes to the stack :

```
0060FE1C 0060FEF0 Stack[00004D58]:0060FEF0
0060FE20 00000001
0060FE24 00000001
0060FE28 00000003
```

The Address at the esp is the address of the array on the stack , and the rest are the arguments.

So we need go back and CALL the function again for the others parts of the array:

We need to insert as a input the command that will make us call the function again,

so we need to use CALL opcode which is E8 and then calculate the offset from the six NOP where the

RIP is = 0x401AB3

To the start of the function we want which located : 0x40197D



Hex value:

401AB3 – 40197D = 136

meaning we need to insert the 0x136 in minus = FE CA

But also in the opposite way in the input: CA FE FF FF

And our input will be : E8 CA FE FF FF

```
80000000
209 2147483648 -80 NULL
The encryption params are 8 (rounds) a
E8 CA FE FF FF
nd 122722333 (delta)
```

#### INPUT:

```
80000000
209 2147483648 -80
NULL
E8 CA FE FF FF
```

#### OUTPUT:

The encryption params are 8 (rounds) and 122722333 (delta)

Goblin:

We used IDA to disassemble the executable safe.exe and after reading it and playing with the input we found out that U, D and C are the only legal inputs and after C we need to provide another input that's a number. After several hours we finally understood the purpose of the program: It's a game of mines! At first the game sets the map where a dot "." Means the tile is clear and X means there's a mine and stepping into it will end the program with exit(1), also F is used for final tiles which the goal is to get to them (there are two) and if a tile has a number on it we can use U or D to jump that number to a new tile, C is used to right at first then left after reaching the maximum count and so on. After that we found a solution (Just by drawing the map and playing the game) and it is:

C  
4  
U  
C  
2  
D  
C  
2  
U  
C  
7

And the output of the program (after successfully finishing) is:

The encryption super secret key is G3f7]0]d]!vRN<86

decrypt.exe:

Next we used IDA to disassemble the executable decrypt.exe and after realizing what the argument should be we ran the program to find the key to the shared safe:

"8" "122722333" "cc5ed245C68AB414" "G3f7]0]d]!vRN<86"

"8" "122722333" "0f0214cbF51A16BC" "G3f7]0]d]!vRN<86"

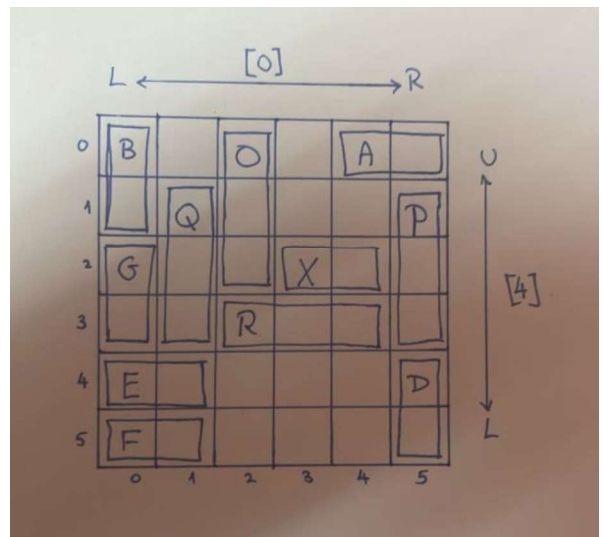
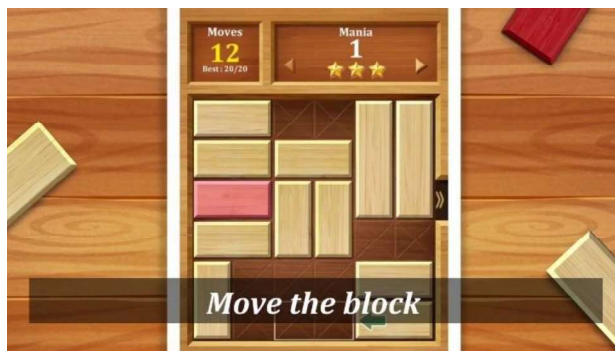
The program is ran twice since block encryption is used and the result is:

QDLZH9NCJRQP2BT2 which is the key to the shared safe.

The shared safe:

We downloaded the zip files.

Immediately ran the analyzer on sheep.jpg a new executable was provided called sheep.out, again we used IDA to disassemble the file and understand it's purpose and after several hours we figured out that it described a game of blocks similar to this:



Each block is assigned a letter as described in the photo, these letter were given by the program so are the initial positions of the blocks, to move a block we would provide it's letter and the directions and steps in that order (for example XL3) the goal is to move X to column 0. The solution to that is:

ER3  
FR3  
QD2  
GD2  
AL1  
PU1  
RR1  
OD3  
XL3

And the result is: ou!amy6u~e game!

Here is a sioqnr6tee code: 1GC8MGJ5IZ. Teg7u6wisely.

We used this code 1GC8MGJ5IZ plus the – B2 from wizard's safe to reveal the board

