# Reverse Hw2:

Tomer Bitan 322218611 Yosef Goren 211515606

**Start:**

- We first download all the files from the vaults and use the analyzer on the images
- We find that the qr has part of a secret message but don't know what to do with it yet.
- In the public safe, we find instructions to crack every other safe to open the shared safe.

**Wizard**:

- At first reading through the wizard, we found the scanf function and located the two places in the code where it's being called assuming these would be the most accessible places to interact with the code and manipulate it.
- Reading through the first one we found that the first pitfall was to get the input after a few permutations to be equal to "DGKPCOEIPCFKNEAMBKLKGFFJOOLK"
    - We figured out that the permutation of our input was: for each char in str_out[i] = (str_in[i]-65 ⊕ str_in[i+1]-65)+65. Where (since the iterations look at string length 28) str_out[27] = (str_in[27]-65 ⊕ str_out[0]-65)+65
    - Since we know the wanted output we can calculate the last char for the input that fits and we can calculate from there each char before it.
    - We found that the correct string was "ADFPACMIAPNICPLLHGMHNLOLCMCJ"
- Once we got this part we saw the next part of the code checks to see if our function's return address is the function's address.
    - We see that there is a call to sprintf right before the check with our original input and the address of the current function.
    - So all we need to do is add more 0s to the end of our previous input to make it so that the appended function address overflows to overwrite the current return address.
    - So our total input is "ADFPACMIAPNICPLLHGMHNLOLCMCJ0000000000000000"
- We get a few letters of the hashed password and a * drawing of the star of david with question marks then the program waits for another input.
- So we move on to analyze the code around the second scanf.
- We see the largest function ever created and with some breakpoint debugging, we can see that there is a loop scanning for 11 values between 0 and 11 that are all different from each other.
- After the numbers from 0-11 in order we see the start of david with numbers on the vertices.
    - Shifting our inputs around results in different variations of the values on the vertices.
    - We find in memory a hardcoded permutation for each imputed value: 0=8,1=6,2=11,3=7,4=4,5=2,6=9,7=10,8=5,9=12,10=1,11=3.

- We noticed a bunch of checkers where every time they sum some of the permutations on the input values and check if they are 26.
- After a lot of reading and digging, we understand that passing all the checks is required to get a print with wAttributes = 2 which represents the color green that indicates success.
- Returning to the sum checks we see the following equations:
  - x1+x2+x3+x4=26
  - x7+x8+x9+x10=26
  - x0+x2+x5+x7=26
  - x0+x3+x6+x10=26
  - x1+x5+x8+x11=26
  - x4+x6+x9+x11=26
  - x0+x1+x4+x7+x9+x10+x11=26
- Putting the equations into a matrix solver we get x6, x8, x9, x10, x11 to be our degrees of freedom and the following simplified equations:
  - x0 = -13 + x8 + x9
  - x1 = -13 + x6 + x9
  - x2 = -26 + x6 + x8 + x9 + x10 + x11
  - x3 = 39 - x6 - x8 - x9 - x10
  - x4 = 26 - x6 - x9 - x11
  - x5 = 39 - x6 - x8 - x9 - x11
  - x7 = 26 - x8 - x9 - x10
- Solving this we get one of many solutions. After doing the reverse repmutation on it we get our desired input: 11 10 2 9 5 0 8 4 3 6 1 7.
- Inputting this indeed solves the puzzle and gives us the hashed password that when contacted with the first letters we get: a81729dc8704D20B3789a53840C12248

**Goblin:**

- We start again with the scarf.
- We analyze the function our input is called with and understand its xored with some constant
- We read through the rest of the code checking the function that calls the function with the scanf because we see the arguments given are used. Doing this we find that the maximum input is 7 steps long and that there are three possible values. C,D,U where C requires another number to be imputed in the same step.
- Eventually, we find that to exit the loop we need to get two values to be 7 at the same time we named them wtc1 and wtc2 (wtc = want to change).
- Inside the loop, we follow two paths. One when the input is C and once when the input is D or U
  - After a bunch of confusion, we figure out that when C is imputed we enter a recursive function that increases wtc2 until it reaches 7 and when it does it increases wtc1 once and starts it iteratively decreases wtc2. When wtc2 reaches 0 it increases wtc1 again and switches back to increasing wtc2.

        - The depth of the recursion is the number that is imputed adjacent to the C and needs to be less than 10.
        - After every iteration there is a check to see if wtc1 or wtc2 are larger than 7 and if some offset + 8*wtc1+wtc2 has an X stored in that location. If so the code exits.
        - Also if the same calculation has F stored there is some other variable that is increased. We called it s (s=somthing)
    - When following the second path we see that when inputting D or U we first check that the same calculation has something that is not a ‘.’ or ‘X’ stored there then:
        - If U is pressed we increase wtc1 by the number stored
        - If D  is pressed we decrease wtc1 by the number stored
        - Then the same check on the value stored and the size of wtc1, wtc2 is performed.
- We try out different sequences of C(number), U, D checking what combination leads us to increase and decrease wtc1 and wtc2 to get 7 in the end.
- Eventually we dive deeper into a function we missed in the start of the start that sets the targets to 7 and we figure out that the memory is meant to be looked at as a grid of 8X8.
    - After flipping it we get:

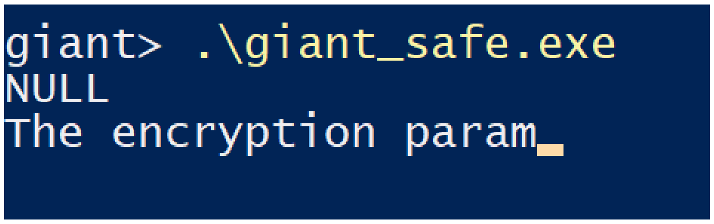
```
. . . . . . . .
F . . . . . . .
X X X X X X X X
. . . . . . . .
. . . . . . . 1
. 4 . . . . . .
X X X X X X X X
. . 7 F . 2 . .
```

    - Once we understand that wtc1 represents the y value and wtc2 represents the x value. And U represents going up when fitting a number and D is down.
    - Now it was much easier to figure out that the input we needed to reach the far side 7,7 was: “C2UC5”
- Now that we got out of the loop we hit a check comparing the s value from before to 2.
    - With the understanding of the map we figured out we need to pass through the both Fs on the way. So we modified our input to be: “C5UC3DC6UC9”
- Doing this gave us the super secret key: “/>*M8%Px7G%Oep(e”

**Giant:**

- Starting with the scan f we noticed three distinct sections to the giants safe:
- The first part was one where a constant expression was defined: 0x4E4C554C, we note that this expression if evaluated byte-wise as ascii is “NULL”, after that there is a scanf that expects 4 chars and then that input is subtracted the constant value - later the subtraction result is dereferenced as a pointer.
- We also note that there was a call to set an exception handler before - which would fix any improper memory access exceptions.

- Our first and correct guess was that we need to intentionally cause the pointer dereference to attempt to evaluate a ‘NULL’ pointer and this way initiate the exception handler and move us to the next step. To do so - all we needed to do was to give the 4 chars ‘NULL’ in the command line:



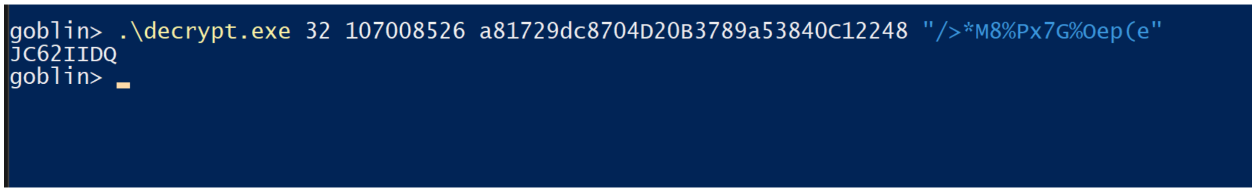
```
giant> .\giant_safe.exe
NULL
The encryption param_
```

- Starting with the second section it required a hex number, then three integers.
    - First the code read the hex number as signed int and negated it if its negative it negates it.
    - The first check after reading the integers required the first integers' least significant byte to be exactly ‘ED’.
        - So our first digit was 237
    - Then the code checked that -57 * input[2]$^2$ - 9120 * input[2] - 364799 = 1
        - Inputting solving this gave us that the third digit is -80
    - The then it checked that the second digit is negative as well as that when we subtract 1 we also get a positive integer.
        - We understand we need to cause an integer overflow so we input -2147483648. Since this is the smallest integer possible decreasing it by 1 will give us a positive.
    - Lastly it checked that the signed version of the hex number has an MSB of 1 (assuming its a negative)
        - So we input 80000000 as our hex number so that it looks like a 0 for the first check but also has MSB of 1 for the second.
- So the input for the second section was: “80000000” then “237 -2147483648 -80”
- For the third section we read through the code and figured out that we need to sort a bunch of constants values. We saw that the values from the scan are given execute privileges with the virtual protect system call and placed on a bunch of nop commands.
    - Since the function is recursive we guessed we need to fill in the commands to finish the merge sort.
    - Indeed there was already one recursive invocation to the function and we know merge sort requires two; moreover the nop hole size was exactly 5 bytes which is the size of the call we needed. So we just needed to find what size jump we need to do - which was the negative of the offset between the hole and the function start: CA FE FF FF (when in little endian).
    - The input we found was: “E8 CA FE FF FF”
- So the total input for the giant was:
    - NULL
    - 80000000
    - 237 -2147483648 -80
    - E8 CA FE FF FF

- Together, all three parts gave us the encryption params: “32 (rounds) and 107008526 (delta)”

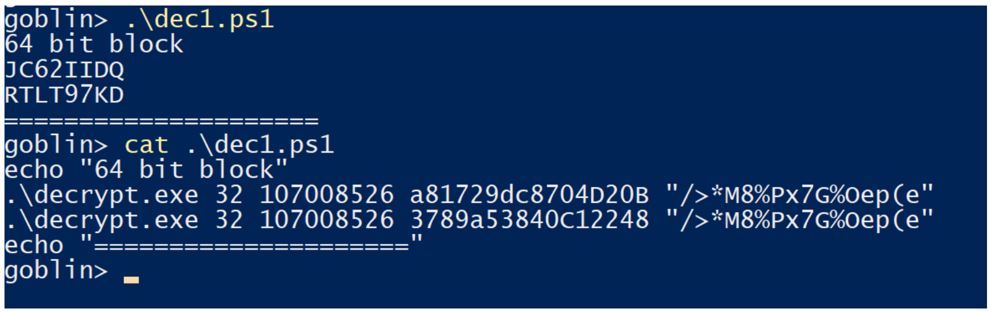
**Decrypt:**

- We started decrypt knowing it expects to get all of the information we have collected in the prior steps. By analyzing it’s disassembly we quickly find a good candidate for the ‘main’ function as it is a function with many submodules that is called directly from the entry point and is followed by a call to ‘exit(0)’.
- In our main candidate we see that the first 5 indices of argv are evaluated (first is cmd name) - meaning it expects 4 command line arguments.
- This makes sense: we have the number of round, the ‘delta’, the hashed password and the encryption key.
- Also we can see that the first submodule call in main is to a function that evaluates the last parameter and ‘throws’ errors about the key having a bad format - so this means that the last argument is the key.
- Hence we have reduces the number of permutations (possibilities for ordering the command line arguments) from 4! = 24 to 3! = 6.
- So we tried all 6 possibilities for ordering the rest of the arguments but only one of these possibilities yielded an output:



```
goblin> .\decrypt.exe 32 107008526 a81729dc8704D20B3789a53840C12248 "/>*M8%Px7G%Oep(e"
JC62IIDQ
goblin>
```

- We were very happy to see a password - but immediately very sad to find out it was the wrong password.
- So we start looking deeper into the code to figure out how the arguments should be given.
- Upon further inspection we see that the hashed password acts as a block of cyphertext to be decrypted - and that the size of the block is 64 bits - which is half of the size we inserted.
- So the next thing to try was to pass the hashed password in multiple batches - one for each block:



```
goblin> .\dec1.ps1
64 bit block
JC62IIDQ
RTLT97KD
=====================
goblin> cat .\dec1.ps1
echo "64 bit block"
.\decrypt.exe 32 107008526 a81729dc8704D20B "/>*M8%Px7G%Oep(e"
.\decrypt.exe 32 107008526 3789a53840C12248 "/>*M8%Px7G%Oep(e"
echo "====================="
goblin>
```

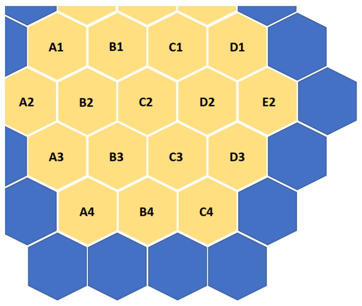
-

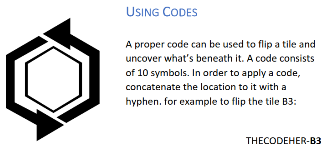
- After splitting the arguments this way can concatenate the results - we got the real password and the safe was opened!

**Sheep:**

- Using the password we got from decrypt in the shared vault gave us a picture and pdf. Using the analyzer on the photo we got a.out file then when changed to an exe gave us code.
- At this point we also take a look at the 'codes.pdf' file we have found and see that this connects with a barcode we have found at the start by running the analyzer tool on 'qr.png': more specifically - 'codes.pdf' specifies how we should use to code if we manage to get it and which tile on the board has which sign. And In the qr we found - we see a slot for the code followed by the identifier of a tile:

_ _ _ _ _ _ _ _ _ _ -B0




USING CODES

A proper code can be used to flip a tile and uncover what's beneath it. A code consists of 10 symbols. In order to apply a code, concatenate the location to it with a hyphen. for example to flip the tile B3:

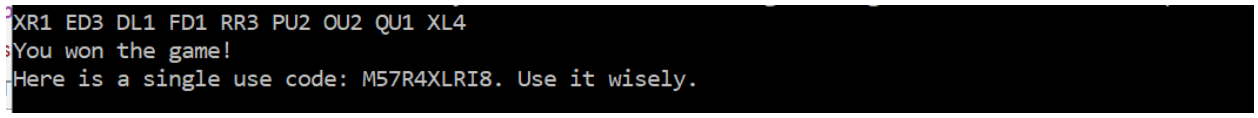
THECODEHER-**B3**

- We opened this with ida and after some digging found the function very similar to the one that credited the board in the goblin.
- We dissect it and rewrite it to look like:

```
+---+---+---+---+---+---+
| R | R | R | . | . | F |
+---+---+---+---+---+---+
| . | . | Q | E | . | F |
+---+---+---+---+---+---+
| P | O | Q | E | D | D |
+---+---+---+---+---+---+
| P | O | Q | X | X | . |
+---+---+---+---+---+---+
| P | O | B | . | C | C |
+---+---+---+---+---+---+
| . | . | B | . | A | A |
+---+---+---+---+---+---+
```

- With some further analysis we understand that we need to choose a car to move: Just like the parking jam game; each letter group is a car - and each car can only move forward and backwards. And there is a specific car that we want to get out of the parking lot by taking it to the edge of it.

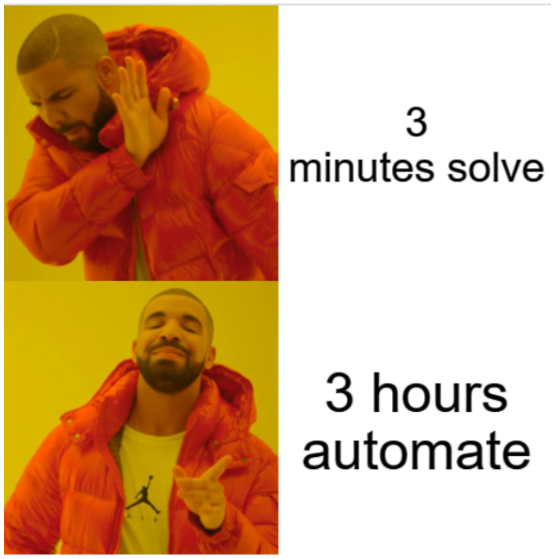
- Additionally - we notice there is a limit on the number of steps that we can take in the game and that limit is 9 moves.
- After further inspection we see that the target car is represented by 'X' - since it can only move right and left - we first attempt to move it left but we see it does not win us the game - so it's likely that we need to move it 3 steps to the left to win the game.
- We find a combination of moves that enables it to move to the left:



```
XR1 ED3 DL1 FD1 RR3 PU2 OU2 QU1 XL4
You won the game!
Here is a single use code: M57R4XLRI8. Use it wisely.
```

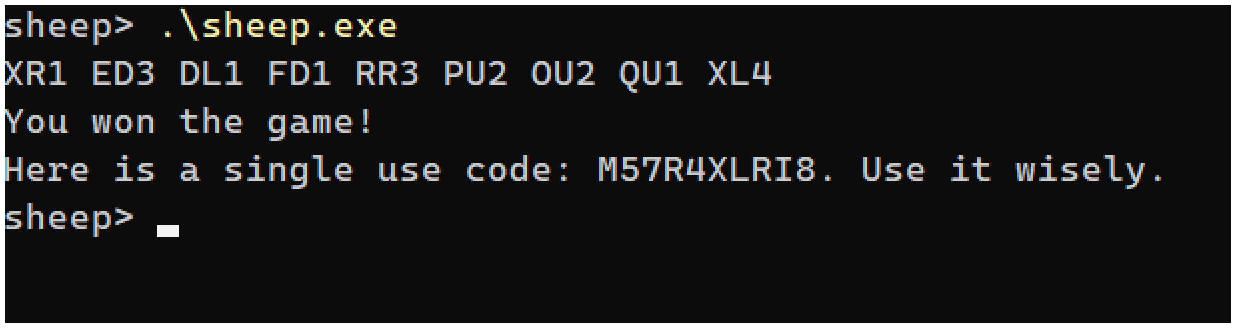
So we are not quite sure if this means we have won the game or not…

- We try using this code to flip the B0 tile - but this does not succeed.
- So we start looking for different solutions.
- At this point we are very tempted to automate the task as is traditional in our line of work:



But the fact we are already 40 minutes late on the submission has made this urge just manageable enough for us to find the solution and move on with our lives.

- Later we manage to find a different solution:



```
sheep> .\sheep.exe
XR1 ED3 DL1 FD1 RR3 PU2 OU2 QU1 XL4
You won the game!
Here is a single use code: M57R4XLRI8. Use it wisely.
sheep> 
```

This one looks more promising…

- So we got to the board on the website once more - and enter the code, and FINALLY we have the sheep:

2
2
6
10
11
9
3
5
11
8
9