Reverse Engineering HW2:

Mori Levinzon 308328467

Omri Klein 318733565

WE CHOOSE TO WORK ON THE FILES RELATED TO THE ID -308328367

We used the passwords we found from HW1 to log to system.

We found out that each player has a directory to folders . There you can find 3 types of fs: private, public and shared(for Giant, Wizard and Goblin) . we downloaded all the files, and we found out there were 3 safe.exe files for each of the following players: Giant, Wizard and Goblin. A quick review in the safe.pdf file showed us that we need to solve each of them in order to open the shared safe and get to the next clues (the sheep files).

Here is a detailed explanation for the way we solved each of them:

Giant:

A brief examination of giant shows that in 3 functions it has scanf calls. We followed each of them in order to find the right input to insert:

The first function set up values in the stack, later by another dll for a subroutine the stack values are changed. The subroutine has a scanf call that expect to receive 5 pairs of hexadecimal code. Later on, there is a call for virtual Protect function which will allow the code the be altered. This is no coincidence and inserting those five hexadecimals words will override the code and replace 5 nop commands with the command that it's opcode is the one we inserted. At first we didn't knew what will be the right answer to write, so we went on the safe bet: we inserted five times 90 (nop in hex) and caused no change to the code.

Upon return from the subroutine we found out that the function iterate over the stack values and print them in some manner from a dll file. The function only printed 4 characters and exited. We understood that the content of the stack caused the program to end and start to further examine the loop. We found couple of things:

1. The stack values must be sorted by size i.e. from the smallest value at the top of the stack to the highest at the bottom of the stack.
2. Subroutine that print characters is called in two loops. The first loop run over the 8 values in the stack one times, the second loop iterates over and over the 8 stack values over and over until we reach some value.
3. In order that the characters printed will make sense, we need that the values in the stack will be :1,2,3,4,5,7,8,9 (in that order).
4. The stack values were swapped from their original placement in the beginning of the function – Caused by the subroutine.

We tracked the stack values trough the subroutine. The subroutine is called recursively which caused the swap. We decided to use given 5 hex values to swap between the stack values in order to get them sorted. A quick look though the web showed us that call opcode is E8 and all the rest is the offset: giving us the first input: E8 CA FE FF FF.

And the output: "The encryption param"

<u>The second function</u> start with setting an exception function (which we will elaborate later) and then we insert 4 characters. Those characters hex values (after swapping the 2nd and 3rd characters) are subtracted from the hex value " 4E4C554Ch" . Later on, we reach extract something from the subtracted value. We found out that if we insert "NULL" the subtracted value is 0 and reaching for that address will cause and exception. By setting the exception at the beginning of the function the program fixed this problem and continued till the end, printing: " s are 64 (rounds) an".

<u>The third function</u> start from receiving a string (with no newlines)

The values are the subtracted from 'A' HEX value to find their relative distance from 'A' (that's were found out that we need to insert ALL CAPS letters) and are cyphered in this manner: every letter distance is XOR with the next letter distance and stored back again as letters with the exception of the last letter which is XORed with the first letter. The result of this cypher is compared to a string in the code and if the first 20 letters are not equals to the stored value we can continue. The following branch is doing a comparison with a stack value we didn't altered in some manner. We tried to alter it's value by causing stack overflow and inserting more characters than intended but the value that was compared was an offset we could insert using any possible pressed letters so we overlowed the buffer so that a address that was placed by sprint call will locate the address specifically where we wanted it to compare it. This adjustment was made by inserting 36 chracters (including the 20 we inserted earlier to the comprasion):

KNALLGPALBKBMOLJIAILKNALLGPALBKBMOLJ

And we going the following output from the followup of the program:

"d 120485897 (delta)"

Wizard:

the 1st function reads first of all one hexadecimal number. But in order to further continue with the program it's need positive and that it's 2 complement will have a SF. The HEX number that follows that limitations is: 80000000.

Later the function reads 3 integers. The first one needs to be 146. The second one needs to be negative (used -1) and the third one need to be 182 later used in the hex file in order that the print will make sense with the start of the printed output.

In the 2nd function as we follow the function we found out that we need to match the values stored in the stack. After we translated the hex values in the stack we found out that we need to insert:

" YwlBu.\srCL7!bdP)E?JG[K hacked this"

In the 3rd' function a Star of David was printed for us and after a brief google search we found out that it's a riddle for a problem that requires to insert 12 numbers from 0 to 11 so that the sum on every edge will be equal after a numerous tries we found out the right answer is: 4 0 1 7 2 11 3 5 8 10 6 9

which further print the output:

The hashed password is: 5832872972622C7325e32c0e51F6FA7E

Goblin

First, we saw that there is no scanf in this file and it does not use any arguments from the command line.
So, we looked for another way to give the file input.
We out that the safe.exe file calls for createPipeLine A CreateFileA. After further reading the MSDN documentation we understood the the safe uses pipes for IO.
We created a script that creates a matching pipeline and connects it to the pipeline opened in the goblin program.

Using the dictionary.txt file we got from the goblin folder we read each line and transmitted through the pipe to the safe file.

The safe chooses one value randomly and transmit answer only to it's value.
That's how we got the output from the safe:

The encryption super secret key is 3Ur(^AH?g86H54#~

Decrypt:

From the previous parts we got:

Wizard – The hashed password is: 5832872972622C7325e32c0e51F6FA7E

Giant – The encryption params are 64 (rounds) and 120485897 (delta)

Goblin – The encryption super secret key is 3Ur(^AH?g86H54#~

After looking at the decrypt file and using the image that was given to us we know that we should put everything we got into the decrypt in order to get the key to the shared safe.

We saw that the order is Giant, Wizard, Goblin.

We put everything in the correct order and got the first half of the key. After that, we saw that the key is not working (it was only half) so we investigated a little more to see that the decrypt file only takes half of the wizard key.

So, we cut the wizard key in 2 halves and put the arguments in the order we found only now we did it two times – one with each half.

We got 2 halves of the key. We put them together and we got the key for the shared folder F7MDIM9CWZ3IQYH9

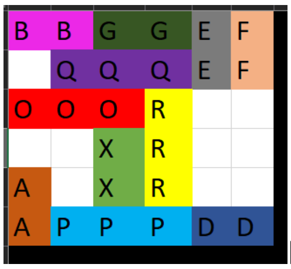
Sheep

We from the shared folder we got Image of some cute sheep and codes file.
We know that the analyzer works on images, so we tried it on the image of the sheep.
The analyzer gave us a binary file – we saved it as an exe file and started to investigate.

We saw that after the scanf there are some conditions that can get up out of the program (inputs we want to avoid).
From the conditions we saw that the first and second characters are chars and the last one is a digit (1-9).

After that we saw some more conditions that showed us that the first character affects the second one in the way that in the memory for each character there is 'H' or 'V'.
On the conditions we saw that if the first letter will give us 'H' we will need to put 'R' of 'L' in order to not exit the program, and if we get 'V' we will need to put 'U', 'D'.
As people that play the computer every now and then, something came up from these conditions:
H-Horizontal = {R-Right, L-Left), V-Vertical={U-Up, D-Down}
We are playing a game.
We also saw that we have only 8 moves to finish the game.

In each move we changed the values of different locations in the memory . each time we changed and tracked after 4 values.
We saw that in the function before the input, some memory is set – a

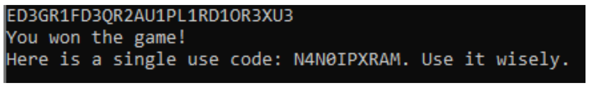
game board.
The game – Rush Hour.
We wrote the board on the side and saw the complete game:


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

Note: the colors are only to make it easier to understand.

We found a solution: [A, Up, 1], [P, Left, 1], [R, Down, 1], [E, Down, 3], [F, Down, 3], [O, Right, 3], [G, Right, 1], [Q, Right, 2].
This did not give us the answer. There are many ways we can reorder these moves and still get the right solution.
We built a program the tries every order that there is until we got the solution.

The solution:



```
ED3GR1FD3QR2AU1PL1RD1OR3XU3
You won the game!
Here is a single use code: N4N0IPXRAM. Use it wisely.
```

We needed to figure out what to do with the code.
On the codes pdf file that we got from the shared folder it says that we can use the code and a name of a tile on the catan board to flip it.
We also remember the we got another image file from the wizard that we did not yet used.
We put the image in the analyzer and we saw that there is a format like there is in the codes pdf with the code missing and the value of the tile we want to flip – C1.

We went on the website, logged into one of the users, navigated to the game board.
We put the code, adding "-C1" at the end and we flipped the sheep tiles!
We finished the assignment! Now we can get all the sheep in the world ,or at least in the game…