Reverse Engineering HW3:

Mori Levinzon 308328467
Omri Klein 318733565

Dry part:

1. In order to change the number of times the function is called, we need to change the value of the frequency written in the global variable. We would hook in the following way: In case we don't have enough nop instructions we'll restore the instructions overwritten by the jump in the start of hook, After that we would change the value stored in the global variable to a small number. We will make sure that we don't change the value again and again each time by using another global variable – Boolean that would be set after the first change to zero, thus preventing from us to change the frequency value again. After the value has changed (or not if it's not our first time jumping to the hook) we would jump back to the rest of the poll function and continue as always.
2. Assuming the function does not change the original value of x during it's run (since the function changes only the local variables on the stack and registers), we would hook the function at it's exit point before the ret: We know x original value (the function parameter), and the return address of the function (saved on the stack), so if we would take the return address and subtract 5 from it we would reach the specific call for the function (call opcode is 5 bytes). In addition, we know that before the ret command esp value must be equal to ebp in order that ret will excute correctly, so if in our hook, we would change the parameter passed to the function (esp+4) from x to x' and the return function address (esp) to address -5, that way we would call the function again, that time with x' as a parameter. In order to not to enter to this function again and again recursively after we end the function we could alter some sort of global variable that will be change to 1 if the hook was used and 0 otherwise. That way we could check every time we enter the hook if the global variable was used and if so we do not alter the address and set to variable to 0 for further use.

3. We'll do the hook in the beginning of the function: In case we don't have enough nop commands we restore the commands overwritten by the jump to the hook. We would read the second argument that suits the string format, use it to calculate the number of parameters, and access each argument in the stack. We can encrypt the format string and each argument on the stack using an encryption function. After that we can go back to the function and send the encrypted message on the socket using the handle parameter.

4. We'll place the hook in the beginning of connect function. In the start of the hook function we would call an auxiliary function which we would write, that would do the following:

1. Get From the stack the return address and use it to calculate parse function address (ret address +the offset in the call instruction) and connect function address the same way (same instruction just for the previous instruction (return address -5)
2. Pop the return address stored by the connect call and save it to global variable elsewhere.
3. Call CreateThread function that will Create a thread that will execute Parse function from the start.
4. Restore the overwritten command from the start of the connect function.
5. Call CreateThread function that will Create a thread that will execute Connect function from the command after the hook jump
6. Call WaitForMultipleObjects in order to wait until both thread for the two functions will finish execution.
7. Push back to the stack the return address from the connect call we popped earlier and saved in another place in the memory.

After the auxiliary function finished running, both functions were executed concurrently , so we will not need to go back to the connect function therefore we'll change the return address from the hook to the instruction after the call for parse by adding 5(call size in bytes) to the return address, that way we well continue to run the program with the next instruction after the call for parse.

5. We would place the hook in the start of the program(in the main/start function):
   First we will search every call for calc and replace it for a call for our auxiliary function-calcWrapper, so that every time calc should be called calcWrapper will be called. And after that we will restore the overwritten instruction.
   Each time calcWrapper is called:
   a. Call Calc (with the same parameters on the stack).
   b. Save the result of the call on the stack
   c. Print the return value from calc function.
   d. Return the result saved in the stack to the caller.

   Using this hook we do not change the calc function so the function that check it's content will pass.

6. a. We would place the hook in the beginning of the function: we would restore the instructions overwritten, change the return address to our code where we would multiply the return value (eax) and then return to the original caller address (saved by the hook in a global variable). If we use the hook as follows then for every recursive call to the function the return value will be multiplied.
   b. we would keep the same hook as before, but with a global variable to keep track of the recursive level: every time the solve is called and we jump to the hook, we first increase the recursive level variable by one and every time we return from solve to the hook, we decrease it's value by one. If the recursive_level value is zero then we are at the outer recursive caller and we multiply the return value, otherwise we keep the return value as is. In order for the hook to keep all the return addresses of all the recursive call we would use

a some sort of second "stack": We would push the return address from the caller each time we enter the hook (as well as increasing the recursive_level value) and pop the return value from the second "stack" and return to this address each time we return from solve (as well as decrease the recursive_level value and multiply it if it equal to 0).

Wet part:

Part 1:

In the first part we were given a keygen exe file that given a password provides site access key, and asked to reverse it's operation. On first glance on the code (through IDA) didn't show us much, but as we run the code we saw that there was a function generated on the stack during runtime and that the code jumped to the function code later on. Through dynamic analysis we found out that each character inserted were printed as another character through some of key-value dictionary form held on the stack. We decided to build an exe file that will reverse the dictionary operation: first we inserted the keygen file all the possible characters in ascii (from 21 hex to 7E hex) and saved the results in a dictionary data structure in reverse from the keygen dictionary structure (the result were saved as the key and the argument character as the value). The reversed dictionary allowed as to build a reverse keygen - given the access key to the site (that we had from the first assignment) we could get the password to the site. And so we did and found the password: FB^!V~*&!^QO/sm7 that opened for us the access to the tools page.

Part 2:

At first glance at client.exe nothing seemed irregular, but once we inserted DMSG in order to get a message from the server we received a uncoherent Gibberish that we couldn't make sense from it. Upon further reading in the second assignment in was noted that the messages from the server are decrypted through secure_pipe.exe. We started to examine the file and found that matter in the message is being decrypted:

For every character in the received message:

1. we transform it to it's hex value.
2. Separate the two characters of the hex value to different instances (for instance 'G' letter is the hex value '47' and we separate it to '4' and '7')
3. For each instance we replace with 1 or 3 characters in the following way:
   a. If char= 'A' print 'J'
   b. If char = 'B' print 'Q'
   c. If char = 'C' print 'K'
   d. If char= '1' print 'A'
   e. If 2<=char<=9 print the char as is.
   f. If char = '0' then get random number and then calculate:
      X = random ^111b +2
      And print X-X.
   g. If char is either 'D','E' or 'F' then do a calculation with a random number : {random/(18-charoffsetFrom0)+ charoffsetFrom0 -9}
      Where we will eventually get number from 4 to 9, calculate it's distance from the character, and print the both the number with '+' separating between them.
      TLDR: we get two numbers that sum is equal the character: For instance if the character F is read we could get the output 9+6 where the sum of both figures is 'F' in HEX.

Given the above conditions, we built a encryption for the message.

In order To make sure that the message will be encrypted in before it's printed to us, we needed to hook the function dynamically:

We searched for the function that get's the message from the server: recv.

We decided that the most suitable hook for this problem is iat hook so we began to look for all the components needed:

1. Upon google search about her we found the dll that it's taken from.
2. 2. We found out the address where the function is been loaded each time and calculated the address offset from the program location.
3. We build the hook in the following way: first, we called recv with the same arguments as given in the stack. After the call we took the second argument, the buffer and run our decryption on it, replacing the encrypted string with our deciphered string.

After that we made sure that the injector will execute with client.exe with DMSG as a given parameter in order to get the function recv called and get the encrypted message.

We run the Injector and got the following message:

I took the robber captive. He is held in B3.If for some reason we should free the guy,one must find a code associated with the ROBBER_CAPTURED event.When this code is used, rolling the dice should result with cubes that sum to seven.Goblin

Part 3:

From the message we figured out that we need to find the exe file that is connected to events in the system.
We look at the description of the files at the website and we found out that codes.exe is related to the events.
We opened the file and looked around. We saw that there are some python commands and to get to them we need to go throw some "strcmp" (comparison) function.
We wanted to find out what are the strings that this function gets every time.
We created an IAT hook on the "strcmp" function so we can see what does it get every time.
"strcmp" is a function that is in the IAT table. That way, it's easier to create an IAT hook for that specific function.
After that, we found out that we can actually change the return value of the function using the hook we just created.
We saw that the first "strcmp" should return with the value of 1 and the second one with a 0.
Our hook function check which time is it (first or second), and then returns the wanted value.
From the message from part 2 we knew that the value of the key is "*ROBBER_CAPTURED*" and because we don't need the second value (we hooked the string comparison), we can just put "*ROBBER_CAPTURED"* in both fields (key_code and old_code).
We upload the necessary files to the website and got the event key: *YFDDXP08D0*

Part 4:

From the message in part 2 we know that in order for the event to happen, we also need that the dices will show the number 7.
In order to achieve that, we need to change the output of the file dice.exe which controls the dices.
We opened the dice.exe file and checked what do we need to happen in order to get 7 in the dices, especially when the "*ROBBER_CAPTURED"* key is entered.
We can see in the code that if the value is 7 (the value that we need), it changes the value so it will not be 7. There isn't a function that we can use to hook.
We need to change the code physically.
We created a small function that checks if the code is "*ROBBER_CAPTURED"* , if

so, it will return 7 and skip the part when we avoid the value 7. If this is not the code, the program will continue normally.
We entered the function at the end of the code, created a jump call from the function that gets the code and returns the number of the dices, to out new function and back to where we want, and don’t forget to restore the lines of code that were where we entered the jump command.
We uploaded the hooked dice.exe file.
After that, we went on the website and entered the code (from part 2) and rolled the dices to get the wanted value – 7 and with it, freeing the robber.