**Reverse Engineering HW3:**

**Mori Levinzon 308328467**

**Omri Klein 318733565**

**Dry part:**

1. .
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:
5. 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 )
6. Pop the return address stored by the connect call and save it to global variable elsewhere.
7. Call CreateThread function that will Create a thread that will execute Parse function from the start.
8. Restore the overwritten command from the start of the connect function.
9. Call CreateThread function that will Create a thread that will execute Connect function from the command after the hook jump
10. Call WaitForMultipleObjects in order to wait until both thread for the two functions will finish execution.
11. 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.

1. 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:

1. Call Calc (with the same parameters on the stack).
2. Save the result of the call on the stack
3. Print the return value from calc function.
4. 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.

1. 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).