__Exercise 01 – Reverse Engineering & Binary Patching__

<u>README.1ST:</u>

1. If you haven't already done so, please make sure to fill out the course questionnaire as soon as possible.
2. Please make sure you have gotten access to your Box.com folder well before you intend to submit your exercise.
3. **Please read the exercise submission guidelines before submitting your solution.**
4. Please provide any tools, scripts, code, IDA idb files or reversing notes.
5. Do not forget to include comments throughout your code **and the code you have analyzed with IDA** (or any other tool), and add at least a short description at the beginning of each file, explaining your thought process.
6. If you use code/data/etc. from an external source – please be clear in mentioning that source in comments / readme file accompanying your code.

<u>Your task, should you choose to accept it:</u>

1. The "message_verifier" program reads a message file given at the command line (i.e.: run "./message_verifier message01")
   Play with it, and try to see what is its output for the different files provided.
   a. Reverse-Engineer the "message_verifier" program, and describe (in pseudo-code) the algorithm that determines whether a particular message will be printed or not.
      Submit as a file named "program_analysis.txt"
   b. Fix the message files, so that all 10 messages will be printed. Please supply all patched messages. Describe your solution in a file named "message_fix.txt".
      <u>Optional:</u> Find two different ways to fix messages.
   c. Patch the binary code (WITHOUT changing the messages themselves), so that ANY message will be printed.
      <u>NOTES:</u> Please explain your patch in a file named "patch.txt", and supply any patch files required, as well as the final patched binary.

< Next question continues on the next page … >

2. The “add_shell_exec” program reads from a file specified at the command line (i.e.: run the command “./add_shell_exec messages” to use the example file named “messages”).
   The program reads and prints each line to stdout.

   a. Your goal is to add new functionality to the program – if a line begins with the characters “#!” (pound + exclamation mark) – you’ll call the system() function on the rest of the line.

      i.e.: for the file:
      Hello
      Meow
      #!cat /etc/passwd
      Goodbye

      The output will be:
      Message 1: Hello
      Message 2: Meow
      [contents of /etc/passwd]
      Message 4: Goodbye

      Note: This is the preferable output, but if you cannot make the original print disappear, the following is also acceptable:
      Message 1: Hello
      Message 2: Meow
      Message 3: #!cat /etc/passwd
      [contents of /etc/passwd]
      Message 4: Goodbye

   b. Using IDA, First, locate the function you’d like to hook – find the code that handles the file, see where you’d like to insert your jump
   c. Locate a good place to add your new code
   d. Decide where your code will return to.
   e. Start building your patch accordingly (using the patch_util_gcc.py script) –
      i. If you’ve chosen the big block of NOPs at the beginning of the function – make the original flow pass over them, by inserting a jmp over them at the start of the block
      ii. First, insert your hook point in the original code – add the jmp to the beginning of your block (right after the jump you added)
      iii. Next, make your code jump back – add a jump at the end of the block of NOPs, back to where you think the code should point

< Question continues on the next page … >

iv. Finally, you are ready to write the actual code (use IDA after each step to verify your results. You may have to use the ‘c’ key to force IDA to parse things as code in some situations) –
    1. Find where the current line is stored
    2. Compare the first two bytes
    3. When appropriate – you’d like to call the system() function
       This program already makes use of the system() function, so it would be easiest to mimic the way it calls system() - including pushing to and restoring the stack after the call, etc.
       To do this, start from the system entry in the 'imports' window in IDA, and use the x-ref function (hit the ‘x’ key on any address/name) to work your way up the chain of function calls, until you find a call that ‘makes sense’ and is easy to integrate with your code (any choice that you can get to work reliably will be considered valid, but try to find an easy solution, not a hard one).
    4. Optional: Decide whether to return to the original flow or act differently (whether you print the command as well or not)

<u>NOTES:</u>

Please submit all patch files, as well as your patched binary