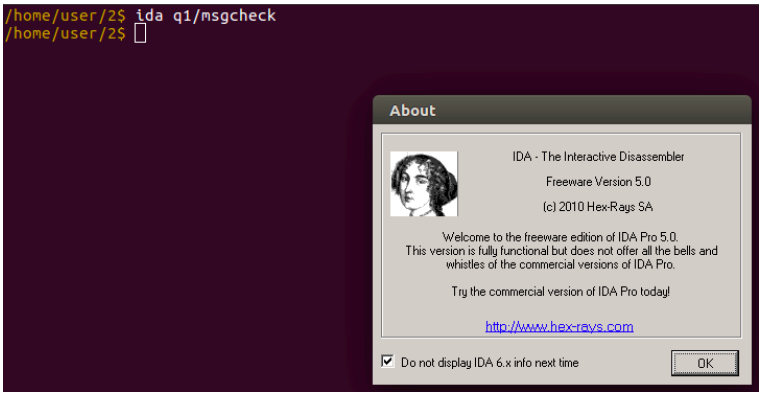
Exercise 2

Log into your VM, open the terminal, and type in `infosec pull 2`: behold `/home/user/2/`!
To run IDA, open the terminal and type in `ida <binary-path>`.



1. **(60 pt)**

   Enter the `q1/` directory. The `msgcheck` program receives a path to a file, and returns `0` if the file is valid and `1` if not; for example, `01.msg` is valid, and `02.msg` is not.

```
/home/user/2/q1$ ./msgcheck 01.msg
valid message
/home/user/2/q1$ echo $? # Check exit code
0
/home/user/2/q1$ ./msgcheck 02.msg
invalid message
/home/user/2/q1$ echo $? # Check exit code
1
/home/user/2/q1$
```

   a. (20 pt)

   Reverse engineer the `msgcheck` program, and write a Python script that works in a similar way. You only need to implement the `check_message(path)` function in `q1a.py`, so that it returns `True` on valid messages and `False` on invalid ones.
   Describe your solution (briefly, and in English!) in `q1a.txt`.

   Note: the `msg` files are in fact six word stories, which is a cool concept in Flash Fiction.

b. (10 pt)

Write a Python script that fixes `.msg` files so that the `msgcheck` program considers them valid. You only need to implement the `fix_message(path)` function in `q1b.py`, so that it writes the fixed message in a file with a similar name and a `.fixed` suffix.

Describe your solution (briefly, and in English!) in `q1b.txt`.

```
/home/user/2/q1$ ./msgcheck 02.msg
invalid message
/home/user/2/q1$ echo $?
1
/home/user/2/q1$ python q1b.py 02.msg
done
/home/user/2/q1$ ./msgcheck 02.msg.fixed
valid message
/home/user/2/q1$ echo $?
0
/home/user/2/q1$
```

c. (10 pt)

Find another way to fix `.msg` files so that the `msgcheck` program considers them valid. Implement `fix_message(path)` in `q1c.py` and describe it in `q1c.txt` like before.

d. (10 pt)

Write a Python script that patches the `msgcheck` program so that it considers all messages valid (i.e., follows the valid code branch whether the message is valid or not). You only need to implement the `patch_program(path)` function in `q1d.py`, so that it writes the patched program in a file with a similar name and a `.patched` suffix.

Describe your solution (briefly, and in English!) in `q1d.txt`.

```
/home/user/2/q1$ ./msgcheck 02.msg
invalid message
/home/user/2/q1$ echo $?
1
/home/user/2/q1$ python q1d.py msgcheck
done
/home/user/2/q1$ chmod +x msgcheck.patched # Make executable
/home/user/2/q1$ ./msgcheck.patched 02.msg
valid message
/home/user/2/q1$ echo $?
0
/home/user/2/q1$
```

e. (10 pt)

Find another way to patch the `msgcheck` program so that it returns `0` for all messages, whether they are valid or not. Implement `patch_program(path)` in `q1e.py` and describe it in `q1e.txt` like before.

```
/home/user/2/q1$ ./msgcheck 02.msg
invalid message
/home/user/2/q1$ echo $?
1
/home/user/2/q1$ python q1e.py msgcheck
done
/home/user/2/q1$ chmod +x msgcheck.patched
/home/user/2/q1$ ./msgcheck.patched 02.msg
invalid message
/home/user/2/q1$ echo $?
0
/home/user/2/q1$
```

2. **(40 pt)**

Enter the `q2/` directory. The `readfile` program reads a file line-by-line and prints it. For example, for `1.txt`:

```
/home/user/2/q2$ ./readfile 1.txt
Line 1
Line 2
#!echo Victory!
Line 3
/home/user/2/q2$
/home/user/2/q2$
```

Write a Python script that patches the `readfile` program so that any line beginning with `#!` is executed *instead* of being printed. For example, for `1.txt`:

```
/home/user/2/q2$ python q2.py readfile
done
/home/user/2/q2$ chmod +x readfile
/home/user/2/q2$ ./readfile.patched 1.txt
Line 1
Line 2
Victory!
Line 3
/home/user/2/q2$
```

You only need to implement the `patch_program(path)` function in `q2.py`, so that it writes the patched program in a file with a similar name and a `.patched` suffix.

Describe your solution (briefly, and in English!) in `q2.txt`.

Since this is a hard question, follow these steps to solve it:

a. Reverse engineer the `readfile` program, and find dead zones into which you can patch your code (they were put there deliberately, so they are a little hard to miss...).

b. One of the dead zones is very small, and is barely enough to redirect to the other dead zone, which is big enough. This means you have to create two patches: write the first (small) one as `patch1.asm`, and the second (big) one as `patch2.asm` in raw Assembly x86, and run the `assemble.py` script on them to translate them to machine code and print them as a Python string.

c. Use IDA to understand into which offset you have to patch this machine code, and to which virtual addresses you have to jump in the code itself.

d. For every line that begins with `#!` call the standard library function `system` (which was, by some blink luck, included in the `readfile` program!), and jump *after* the `printf`; for every line that doesn't, jump *before* the `printf`.

Note: instead of repeatedly calling `assemble.py` on `patch1.asm` and `patch2.asm`, and then manually copying and pasting this machine code into `q2.py`, just `import assemble` in `q2.py` and call `assemble.assemble_data` or `assemble.assemble_file`.

A few general notes:

1. Document your code.
2. Don't use any 3rd-party libraries or Python packages the grader wouldn't have.
3. If your answer takes an entire page, you probably misunderstood the question.
4. Each student gets slightly different binaries, so don't be surprised if your solution isn't the same as other students'.
5. The grader is still dealing with existential angst, so funny comments are still welcome.