Password Protection for xv6

In this assignment you will make xv6 the most secure toy operating system around by requiring a password to log in.

Getting the Code from Github

As usual, we'll be working off of a slightly modified version of xv6. This version includes an implementation of the `bcrypt` algorithm, which is an intentionally slow password hashing algorithm. It also includes a new system call, `random()`, which generates a random unsigned 32-bit integer.

If you still have your xv6 directory from last time, remove or rename it first. Then get the base xv6 code for this assignment:

```
$ git clone https://github.com/moyix/xv6-public.git
Cloning into 'xv6-public'...
remote: Counting objects: 4562, done.
remote: Compressing objects: 100% (45/45), done.
remote: Total 4562 (delta 14), reused 0 (delta 0), pack-reused 4517
Receiving objects: 100% (4562/4562), 11.70 MiB | 3.96 MiB/s, done.
Resolving deltas: 100% (1839/1839), done.
Checking connectivity... done.
$ cd xv6-public/
$ git checkout hw8
Branch hw8 set up to track remote branch hw8 from origin.
Switched to a new branch 'hw8'
```

Password Authentication

When a user firsts boots xv6, we will assume there is no password set. You should implement code that checks if a password file exists, and if it does not, prompts the user to enter a password. Because users sometimes mistype their passwords, you should ask for the password twice, and then compare them to make sure they match.

Once the user has entered a password, you should:

1. Generate a random *salt* value.
2. Hash the users's password using the `bcrypt()` function.
3. Write the salt and the hash out to a file.

To verify a password:

1. Ask the user to enter a password.
2. Read the salt and hash from the password file.
3. Use `bcrypt_checkpass()` function to see if the password is correct.

A typical session might look like:

```
cpu1: starting
cpu0: starting
sb: size 2000 nblocks 1941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start
58
init: starting sh
No password set. Please choose one.
Enter password: test
Re-enter to confirm: tast
Passwords do not match. Try again.
Enter password: test
Re-enter to confirm: test
Password successfully set. You may now use it to log in.
Enter password: test
Password correct, logging you in.
$
```

Hints

- You should not let the user run any commands until they have entered the correct password. So, a reasonable place to put your code is inside of `init.c`, before the shell is started.
- You can use the `random()` function to get a random 32-bit integer for the salt. Because the salt is 16 bytes (128 bits), you will have to get multiple random integers and use them to fill out the salt value.
- Look at `bcrypt.h` for documentation on the `bcrypt` and `bcrypt_checkapss` functions.

Submitting

As usual, you will use git to create a patch.

Commit your changes:

```
git commit --all --message="Add password protection"
[hw8 43f5a6a] Add password protection
 1 file changed, 114 insertions(+)
```

(Note: if you added any new files, you will also have to use `git add <filename>` before you run `git commit`.)

Now create the patch file:

```
$ git format-patch hw8.unmodified
0001-Add-password-protection.patch
```

The command creates a file, `0001-Add-password-protection.patch`, containing the changes you've made. Submit this file on NYU Classes, along with a `partner.txt` listing your partner (if any).

Submission Notes

- Non-compiling code will not be graded. Be *sure* that your code compiles before submitting it!
- Don't try to edit the patch file after creating it. Doing so will most likely corrupt the patch file and make it impossible to apply. Instead, change the original file, commit your changes, and run `git format-patch` again. Then submit *both* patch files.

Security Notes

The implementation you will create is not bad, but does have some weaknesses:

1. The random number generator used is very weak, and depends on the current time. This means that the range of possible salts in practice is much smaller than it could be. This might allow someone to precompute the hashes for common passwords.
2. An attacker who has access to the filesystem image (`fs.img`) could modify the password hash on disk or delete the password file, allowing them to log in. On a real machine, however, this would be more difficult (since the attacker would have to take the hard drive out of the machine first).