Tales from the `crypt`

CMS 495, Spring 2016

Due Thursday, March 18, at 11:59 PM

Description

In this project, you'll write your own dictionary-based password cracker using the C `crypt` utility.

Your cracking program will take as its input a shadow password file consisting of usernames and their associated password hashes, along with a list of candidate passwords.

The program should determine the true password for each user by calculating the hash of each entry in the candidate file, then comparing those hashes to the entries stored in the shadow password file. In addition, your cracker must include some support for *mangling* the candidate passwords to deal with more potential matches.

The project zip file contains the following:

- This write-up
- `cryptExample.c`, which illustrates how to use `crypt` to calculate hashes
- `password.lst`, the (slightly edited) list of popular passwords distributed with John the Ripper
- `shadow`, the first shadow file consisting of 100 plain hashed passwords from the word list
- `shadowMangled`, a second file with 100 mangled passwords
- `generateShadowFiles.py`, a Python script for creating test shadow files from `password.lst`

Compiling, Running and Grading

Your program should be named `crack.c` and housed in a `Project3` directory on Cloud9. Include a `Makefile`—make sure `make` actually works!

Your program should run at the command line with the following format:

```
shell$ ./crack -i shadow -o shadow_cracked -l password.lst -m
```

The `-i` flag specifies input file and `-o` specifies the output file, which will hold the usernames and their associated cracked passwords. The `-l` flag specifeds the word list file. The `-m` is optional; if it's present, your program should incorporate the mangling rules into its attack.

Use the `getopt` function to process the command line arguments. It can automatically scan the `argv` array and pull out each flag and its associated value. This is better than assuming the arguments are in any fixed order.

I will grade your code by running it against both shadow files and validating the cracked passwords it produces. I will also test it against one more file of 100 passwords generated by the same Python script as the two examples.

The Shadow Files

Each shadow file entry has the following basic format

```
username:hashvalue:truepassword
```

Your program should read the hash value from each line and reconstruct the true password using the dictionary. The true password is provided so you can check your answer but you **MAY NOT** use it as part of the cracking process.

You can use `strtok` or `strsep` to parse the entries on each line.

Both files are generated by `generateShadowFiles.py`. You can run the program to create additional test files:

```
shell$ python generateShadowFiles.py -o test -s 0 -m
```

The `-o` flag specifies the output file name, `-s` sets the seed of the random number generator and `-m` (if it is present) mangles each password before hashing it.

C Password Hashing

The `crypt` function calculates DES password hashes. Its signature is

```
char * crypt(char *key, char *salt)
```

The first input string is the password.

The second string is the salt value. In this project, the salt string should always be `"$1$"`.

The function returns a string representation of the DES hash of the input key with the given salt.

To use `crypt` you must include `<crypt.h>` and `<unistd.h>`. You must also link in the `crypt` library by including the flag `-lcrypt` in your `gcc` command.

Mangling

If the `-m` option is given on the command line, `crack` should add mangling rules to each password that it checks.

You only need to support one mangling rule: appending each single digit 0-9 to the candidate password. The program should test the unmodified password even if mangling is activated.