Tales from the `crypt`

CMS 230, Fall 2017

Due Friday, 12/15, 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 dictionary 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.

Shadow Password Files

You might remember that we worked with password files in one of our early labs. Take a moment to review that. I'll wait.

Okay, welcome back.

Recall that a Linux system does not store users' raw passwords. Instead, it stores the **hash** of each password. When a user wants to log in or perform any operation that requires password validation, the system calculates the hash of the submitted password and compares it to the stored hash in the shadow file. If the hashes match, then the user has, with high probability, submitted the true password.

Note that this depends on having a strong cryptographic hash function that is one-way and has a low probability of generating collisions in its output space.

The shadow password file has entries like this, one per line:

```
username:hash
```

Each shadow file in the `tests` directory has five entries. The usernames are dummies—you won't use them at all in this project.

Each test shadow file has an associated truth file that gives the real passwords for each hash. The grading script uses these to check the output of your program. **Your cracking program cannot make any use of the truth files**, because, duh.

Tokenizing the Input String

After reading each line from the shadow file, you'll need to separate the hash string from the user name. This can be done with the `strtok` function, which tokenizes a string, splitting it into parts based on a delimiter character.

`strtok` takes a string and a list of delimiters as inputs, then returns the substring up to the first appearance of a delimiter. Subsequent calls return further substrings, splitting on the specified delimiters. Here's an example:

```
FILE *f = fopen(inputFile, "r");

char buffer[128];

while (fgets(buffer, sizeof(buffer), f)) {
    char *user = strtok(buffer, ":");

    // Use NULL to extract the next token from the previous input
    char *hash = strtok(NULL, ":");

    // Strip the terminating newline if it's present
    if (hash[strlen(hash) - 1] == '\n') {
        hash[strlen(hash) - 1] = '\0';
    }

    // Dictionary search for matching hash
    crack(hash);
}
```

Notice the section that strips the terminating newline. Remember that `fgets` returns the newline, but it isn't part of the hash value. You'll also need to strip newlines when you read from the dictionary file.

Crypt

You will use the C `crypt` function to calculate password hashes. Take a look at `example.c` to see how it works.

`crypt` takes two arguments: the string to hash and a ”salt” string. The salt string serves two purposes:

- It begins with a number that identifies the hash function. 1 corresponds to the `md5` message digest function, which is not the most secure option, but is fast. 6 is SHA-512, a more realistic choice.
- It can contain additional characters that are added to the input string before the hash operation. This operation increases the length of the input string, defeating attacks based on precomputing hash values for short passwords. Read about the ”Rainbow Table Attack” if you’re interested in learning more.

Your salt string will always be `$1$`.

Command-Line Options

Your program must support the following options:

- `-i`: specifies the input shadow password file.
- `-o`: specifies the output file that will store the cracked passwords
- `-d`: specifies the dictionary file that stores the list of candidate passwords. The supplied `words` file is a UNIX dictionary with a large number of words intended for spellchecking.

The basic dictionary attack simply calculates the hashes of every word in the dictionary, checking for matches with the entries in the shadow file. A more sophisticated attack supports ”mangling” the dictionary words to generate more candidate passwords. Your program must support three mangling options.

- `-n`: append a single digit, 0-9, to the end of the candidate.
- `-c`: toggle the case of the first letter of the candidate.
- `-l`: apply `leet`-style mangling rules to the **lowercase** letters of the candidate. The rule should implement the following substitutions:
  - `a`→`@` (at)
  - `e`→`3`
  - `i`→`!`
  - `l` (letter l)→`1` (one)
  - `o` (letter o)→`0` (zero)
  - `s`→`5`

All substitutions are applied to the candidate word. Note that uppercase letters are not affected. For consistency, **the case toggle option has precedence over this one**.

These options can be invoked in any combination. It's acceptable to use zero, only one, any two, or all three mangling options, but you can only invoke one combination of options each time the program runs.

You must use `getopt` to process the command-line options.

Compiling, Running and Grading

Your program should be named `crack.c`. Include a `Makefile` to build your program from source. Submit your program to GitHub by the due date.

Test your code using the Python script `test.py`.

```
prompt$ python test.py.
```

Your grade will be the fraction of tests you pass.