CSE111 Spring 2022

Assignment 6

In this lab you will implement a distributed, multi-threaded password cracker with complimentary functional and non-functional requirements.

This assignment is worth 14% of your final grade.

Late submissions will not be graded.

Background information

The standard Unix password one-way hashing function `crypt()` can be used to crack passwords simply by trying every combination of the 95 characters on a standard US keyboard. However, the search space for such an attack is enormous as there are 6,634,204,312,890,625 (95^8) possible passwords.

Even for short passwords, it is computationally expensive to undertake such an attack unless it can be parallelised and distributed. In this assignment you will develop such a distributed, parallel password cracker using techniques developed in earlier assignments.

Setup

SSH into any of the CSE111 teaching servers using your CruzID Blue credentials:

```
$ ssh <cruzid>@<server>.soe.ucsc.edu
```

(use Putty http://www.putty.org/ if on Windows)

Remember there are FOUR teaching severs available for this assignment, you will use them all:

```
noggin.soe.ucsc.edu          olaf.soe.ucsc.edu
nogbad.soe.ucsc.edu          thor.soe.ucsc.edu
```

Create a suitable place to work: ***(only do this the first time you log in)***

```
$ mkdir -p ~/CSE111/Assignment6
$ cd ~/ CSE111/Assignment6
```

Install the lab environment: ***(only do this once)***

```
$ tar xvf /var/classes/CSE111/Assignment6.tar.gz
```

Build the skeleton system:

```
$ make
```

Requirements

Basic:

Listen for a list of password hashes multicast on the teaching server private network
Decrypt password hashes back to their four-character plain text equivalent
Return these plain text passwords to a TCP server nominated by the multicast server

Advanced:

As basic; but do so as quickly as possible.
Simple solutions will be embarrassingly parallel
Better solutions will be truly parallel
The best solutions will be truly parallel and distributed

These requirements are NOT equally weighted - see Grading Scheme below.

What you need to do

You need to develop a password cracker that does the following:

- Listens for encrypted passwords sent to it via UDP unicast
- Cracks the passwords
- Sends the cracked passwords back to whatever server sent them to you via TCP

There are several possible implementations, it is recommended you start with the simplest then work your way up to the more complex solutions:

Single Server / Single Threaded

- Received encrypted passwords are cracked one after the other

Single Server / Embarrassingly Parallel

- Received encrypted passwords are cracked in their own thread

Single Server / Truly Parallel

- Received encrypted passwords are cracked in as many threads as you have CPU cores (24)
    - If you have more cores than passwords, this is a good solution
    - If you have more passwords than cores, it adds little

Multi Server / Embarrassingly Parallel (you may choose to skip this one)

- Received encrypted passwords are cracked in their own threads on multiple machines
    - If you have many passwords to crack, this is a reasonable solution
    - If you only have a few passwords, you're wasting processing capacity

Multi Server / Truly Parallel

- Received encrypted passwords are cracked in as many threads as you have CPU cores across all the machines available to you (96)
    - Best solution as it makes use of all cores regardless of how few / many passwords need cracking

You also need to develop a simple test server that sends the multicast to your password cracker. This server is purely for you own testing, eventually you will need to respond to multicasts coming from the grading system. See below for details.

Password cracking

The following single-threaded function is provided to crack the four-character passwords you will receive from the test server:

```
void crack(const char *alphabet, const char *hash, char *passwd);
```

See `cracker.h` for details.

Simple, single threaded or embarrassingly parallel solutions can use this function as-is. More sophisticated, truly parallel solutions will need to implement their own, multi-threaded password cracker.

Sample passwords and hashes

The following four-character passwords and matching hashes can be used for testing:

Hash	**Password**
WuxaoJbgzwuwQ	ucsc
Uz8uY1RkTJzLs	BSOE
54rHK/8o0XZ5Y	UCSC
3Wa0jNDoijFFk	5150
YNNbxoodYqWaU	2to1

Listening on the correct multicast address and port

You should only listen on your CruzID specific multicast address and port, which can be determined by calling the following functions:

```
in_addr_t get_multicast_address();
unsigned int get_multicast_port();
```

See `cracker.h` for details.

To use it in your code, you should do something like this:

```
struct sockaddr_in server_addr;
bzero((char *) &server_addr, sizeof(server_addr));
server_addr.sin_family = AF_INET;
server_addr.sin_port = htons(get_multicast_port());

if (bind(sockfd, (struct sockaddr *) &server_addr, sizeof(server_addr)) < 0)
    exit(-1);

struct ip_mreq multicastRequest;
multicastRequest.imr_multiaddr.s_addr = get_multicast_address();
multicastRequest.imr_interface.s_addr = htonl(INADDR_ANY);
if (setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
          (void *) &multicastRequest, sizeof(multicastRequest)) < 0)
    exit(-1);
```

Your test server should multicast on the above address and port, and should listen for the returned, cracked passwords on your CruzID specific unicast port which can be determined by calling the following function:

```
unsigned int get_unicast_port();
```

See `cracker.h` for details.

How your cracker will be invoked

The automated grading environment will one instance of your `./cracker` on each of the four teaching servers. If you have a single server solution, you code needs to be aware of which server it is running on so three of them do nothing.

To retrieve the hostname of the teaching server you are running on into `hostname` buffer:

```
char hostname[64];
gethostname(hostname, sizeof(hostname));
```

Schedule

This assignment is in five stages:

Stage 1: **Development**: Starts 19:00 Tuesday May 24 2022

Develop your password cracker, testing as you go. You will need to generate the password hashes and multicast packets yourself. It is your responsibility to ensure your tests are comprehensive.

Stage 2: **Test**: Starts 10:30 Monday May 30 2022

A test server will multicast CruzID specific datagrams at regular intervals. All CruzIDs will receive their datagrams at the same time. You should ensure you are receiving these datagrams correctly and are returning the cracked passwords to the correct server. A webpage will be made available so you can check the accuracy of your solution; see below for details.

Stage 3: **Practice**: Starts 08:00 Thursday June 2 2022

The test server will switch to sending single CruzID specific datagrams at 30 second intervals. There are 160 students in the class so you will receive a personal datagram once every 80 minutes

Stage 4: **Evaluation**: Friday June 3 2022

You lose access to the teaching servers at midnight on Thursday June 2. Your code will be run for you by the automated grading system.

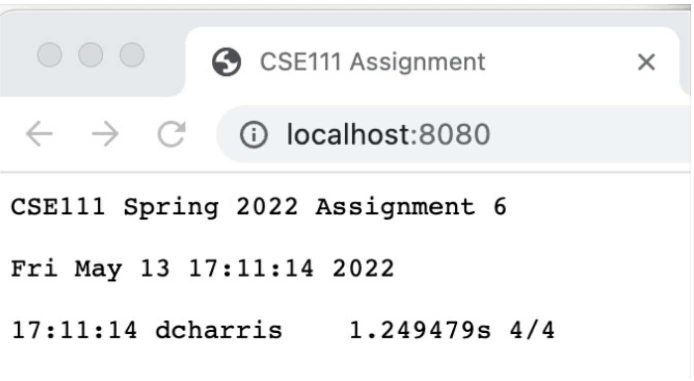
Accessing the Results Page

You'll need to create an SSH tunnel to the web server displaying the results.

If you're on a Mac or other UNIX-like operating system, type the following command in a terminal and enter your Blue CruzID password:

```
$ ssh -L 8080:nooka:8080 <cruzid>@<server>.soe.ucsc.edu
```

Then go to the http://localhost:8080 in a browser:



```
CSE111 Spring 2022 Assignment 6

Fri May 13 17:11:14 2022

17:11:14 dcharris    1.249479s 4/4
```

If you are cracking passwords, successfully or not, and sending them back to the server, your cruzid will be on this page somewhere. The page refreshes automatically.

Note: If you are on a Windows machine, this page may help to get the SSH tunnel setup. If it doesn't, remember Google is your friend: https://phoenixnap.com/kb/ssh-port-forwarding

Grading Scheme

The following aspects will be assessed:

1. (20%) **Does it work?**

   a. All passwords accurately decrypted (single server, single threaded) (20%)

   Marks deducted pro-rata for any incorrectly decrypted passwords

2. (70%) **Is it fast?**

a)	All passwords decrypted in <	100 seconds	(single server, embarrassingly parallel)	(10%)
b)	All passwords decrypted in <	50 seconds	(multi server, embarrassingly parallel)	(30%)
c)	All passwords decrypted in <	25 seconds	(multi server, truly parallel)	(60%)
d)	All passwords decrypted in <	10 seconds	(multi server, truly parallel)	(70%)

3. (10%) **Is it right?**

   a. Your implementation is free of compiler warnings and memory errors (10%)

4. (-100%) **Code fails to compile.**

   Whilst your code will not be executed (the principal assessment taking place "live" on the teaching servers) it must compile on the grading system. Take care to avoid using 3rd party libraries that will not be available on the grading system.

5. (-100%) **Did you give credit where credit is due?**

   a. Your submission is found to contain code segments copied from on-line resources and you failed to give clear and unambiguous credit to the original author(s) in your source code (-100%)

   b. Your submission is determined to be a copy of another student's submission (-100%)

   c. Your submission is found to contain code segments copied from on-line resources that you did give a clear an unambiguous credit to in your source code, but the copied code constitutes too significant a percentage of your submission:

< 25% copied code	No deduction
25% to 50% copied code	(-50%)
> 50%	(-100%)

What to submit

In a command prompt:

```
$ cd ~/CSE111/Assignment6
$ make submit
```

This creates a gzipped tar archive named `CSE111-Assignment6.tar.gz` in your home directory and checks it will execute successfully in the automated grading system.

When you're happy that your submission is working as expected:

UPLOAD THIS FILE TO THE APPROPRIATE CANVAS ASSIGNMENT.