Computer Networks & Distributed Processing (Spring 2014)
Project 3: Distributed Password Cracker

Due: Friday July 11, 2014 @ 11:59pm

For several decades, UNIX-based systems (and now OS X and windows) all store only the hash of a user's password. When you type your password to log on, the system hashes the password you type and compares it against the stored hash value. So, an attacker who takes over a computer and steals its `/etc/passwd` file[1] doesn't immediately learn the cleartext passwords of the users.

As we saw in class, a cryptographic hash function is a hopefully one-way transformation from an input string to an output string, where it's (hopefully) impossible to go backwards from the output to an input that will generate the output:

$$\texttt{crypto-hash}(\textit{cleartext}) \rightarrow \textit{hash value}$$

This design was a great start, but it's not unbreakable. The original, and still somewhat commonly used, way of hashing passwords uses a hash function called `crypt()`. When initially designed in 1976, `crypt` could only hash four passwords per second. For this reason, it was determined that `crypt` only looks passwords up to 8 characters[2]—longer passwords are truncated.

The situation has changed, however: with today's drastically faster computers, incredibly optimized password cracking code can attempt over millions of hashes per second on a single core. While the hash function is still one-way, you can find a lot of passwords using a brute-force approach: generate a string, hash it, and see if the result matches the hash of the password you're trying to find. Still, a few millions of hashes per second isn't all that fast. There are $62^8 \approx 2.18 \times 10^{14}$ possible eight character passwords using only the upper, lowercase, and numeric characters. This would take 842 days to crack using a single core. **But why stop there?** when there must be a few hundred idle cores sitting around campus...

In this project, you will create a distributed password cracker. You will mix and match the concepts of concurrency, threading, and client/server communication protocols (and RPC if you wish) to solve this task as quickly as possible. Your goal is to create a distributed system that can run across the entire Internet. This is an "embarrassingly parallel" application—it consists of a set of expensive operations on small chunks of data, but there's no data that needs to be shared between nodes. They just receive a work unit allocation, try all of the passwords in that unit, and tell the server if any of them was a match.

You will write: a job dispatcher server, a worker program, and a end-user client (details below).

1 Logistics

- For this project, you can work individually or in a team of two.[3] You can discuss ideas/design in general terms with other people taking the class, not just your project partner. You may help debug your friends code—though, this must not mean sharing code with another team.

[1] This is where the hashed passwords are kept in the old days

[2] There are potentially other "peculiarities" of `crypt` waiting for you to uncover and use to speed up your code.

[3] Unless you really like working by yourself, it's generally good to have a project partner.

- This project is to be implemented in Python, Go, C/C++. There will be a demo day, and there will be a speed contest (details to follow), so your choice of language might determine how fast/scalable your code will be.
- Use of a version-control system (VCS) such as Git, Mercurial, SVN, etc. is strongly encouraged. If you need a place to sync your repository, BitBucket (`www.bitbucket.org`) is a reasonable option, which is currently free for a small team.
- You'll hand in a tar ball or a zip file named `p3.{tar|tgz|zip}`. When you're ready but before the deadline, upload this file to Canvas under the assignment section.

  Be sure to include a README file with instructions on how to operate your program. If you're using Go, include in the README how to build your binary.

  Importantly, this README file will describe and document your design (if someone wants to pick up your program, s/he should be able to understand how it works by reading this and the comments). Did I mention, comment your code?

1.1 Basic Requirements

Your password cracker consists of three programs: the worker, the end-user (request) client and the (dispatcher) server. A request client sends a password-cracking job to the server, and the server is responsible for dividing the job into parts and allocating those parts to worker clients. The clients and server communicate with each other using a protocol/design of your choice. (We recommend UDP packets.)

The server must be robust to communication failures (clients never receive a message) and to client failures by eventually assigning uncompleted jobs to other clients. It is your responsibility to ensure that your code is well-tested.

If Control-C is pressed at the server, it should shut down all the client workers before quitting.

1.2 The `crypt` function

UNIX's `crypt` function has a great man page (try `man 3 crypt` on your command-line). It takes two arguments: the key and a "salt" value. In Python, you can access the `crypt` function by importing `crypt` and calling the function `crypt.crypt`. Alternatively, you could say `from crypt import crypt` and call `crypt` directly.

The key is the password you want to encrypt. The salt is a constant string that can be used to produce different encrypted versions of the same key. That is:

```
crypt("hello", "aa") -> "aaPwJ9XL9Y99E"
crypt("hello", "ab") -> "abl0JrMf6tlhw"
```

You'll note that the salt appears in the hash value. The purpose of the salt is to deter precomputed dictionary attacks in which an attacker, in advance, computes the hash of tons and tons of strings and then simply looks up the result.

For this assignment, you'll always use the salt "`ic`".

1.3 Your Dispatcher Server

The server should recognize the two different clients as they contact it. It should keep track of the clients and which jobs they're currently working on or what job they've requested the server to solve. It should handle timeouts.

It's important that the server be concurrent. The server may allocate jobs as it sees fit. It is your job to ensure that the jobs are of reasonable size and that the load is balanced across the worker clients. The server should do this by having the number of workers assigned to each request be as equal as possible, without preempting workers. For example, there are 10 workers, which are all working on one request, and a new request comes in. As the workers finish their jobs, they are assigned to work on the new request, until each request is being handled by 5 workers.

Your code should will use the following command-line arguments: `./server <port>`

Example: `./server 2222`

1.4 Your Workers

The worker client should also be concurrent to facilitate communication with the server. It is probably easiest to use multithreading: one thread doing the actual work of the client (the cracking) and a second thread, listening for messages from the server. The clients should be independent and should be able to find out everything it needs to know through the protocol you'll design. The client should not remember jobs it's worked on in the past.

Your worker will be invoked using the following command line arguments:

```
./worker_client <server hostname> <port>
```

Example: `./worker_client server.world.com 2222`

1.5 Your End-User Client

The request client does not need to be multithreaded, but it certainly can be if you like. Its job is to request that a password be cracked by sending a hash to the server. It is also required to ping the server every 5 seconds to remind the server of its existence. Since we do not know how long it will take the server to find the answer, this approach is more feasible than defining an arbitrary timeout value.

Your end-user requester client should use the following command-line arguments:

`./request_client <server hostname> <port> <hash>`

Example: `./request_client server.world.com 2222 aaHLWHfiLg`

2 Technical Notes & Tips

- Though this is completely optional, it usually helps to start out small and grow out to a full-featured system. In this case, you might want to begin with a local password cracker. Implement a test client that just runs from the command line. If your executable is called cracker, it will accept a range of characters and a hash value and attempt to crack the hash using all strings in the range of characters, like so: `./cracker <begin> <end> <hash>`

  For example, `./cracker AAAAA ABAAA sdfj325SWE` It should output NO if the password is not within the specified range and should output YES if it is.

  For ease, use the ordering for the range as follows: A B ... Z a b ... z 0... 9

  This means that if the range is AAAAA ABAAA, your code should check:

  ```
  AAAAA
  AAAAB
  ...
  AAAAZ
  ```

```
AAAAa
AAAAb
...
AAAAz
AAAA0
AAAA1
...
AAAA9
AAABA
AAABB
...
ABAAA
```

- Your remote workers may become unavailable temporarily or permanently. Because you're harvesting the work of "volunteer" computers, it is entirely possible that the nodes may run at widely disparate speeds. Some may be highend servers, others may be cheap ten-year-old desktops. You'll have to make sure that your server properly balances the load across these nodes so that everybodys busy, but so that you dont accidentally allocate 10 years of work to an ancient 486!
- At minimal, the following messages are suggested for communicating among the server, workers, and requesting client:
  - `request_to_join` - a worker requests to join the dispatcher server.
  - `assign` - the server assigns a job to a worker. This probably should look like
    `assign AAA ABA a82kgjad`
  - `ack_job` - the worker acknowledges the receipt of a job.
  - `ping` - used for request clients to ping the server, and for the server to ping workers.
  - `done_none_found` - used by the worker to signal the server that it has completed the assignment without finding the password; also used by the server to communicate with the requesting client.
  - `done_found` - used by the worker to inform the server that it has completed its job and has found the password, and by the server to tell a requesting client the password.
  - `not_done` - when a worker client has received a `ping`, but is not done processing its job, it sends this command to the server.
  - `shutdown` - when the server is asked to terminate, it sends every worker client this command.
  - `crack` - when a requesting client wishes to make a request, it sends this command to the server with the hash to be cracked.
- Failures *will* happen. What's a reasonable behavior when a worker doesn't acknowledge the receipt of a job? How about the worker failing to report back?

  We recommend a design where the requesting client sends sending a `ping` message (not your system ICMP ping) to the server every 5 seconds. The server should monitor these pings and if it does not receive a ping from a request client in 15 seconds, the server should assume that the request client died, and remove its request from the queue.

2.1 Note for the curious

`crypt` is not a recommended hash function to use today for most applications. Many implementations of crypt have a deadly flaw when used outside of simple UNIX passwords: they will discard characters after the 8th! Those curious about the use of hash functions for applications such as Web security should take a look at the paper "Dos and Don'ts of Client Authentication on the Web", which discusses these vulnerabilities and best practices in more detail. For more general cryptographic hashing, consult the Web—there are several alternatives depending on your security and speed needs and the way you want to use the hash function. But don't use crypt!