

HW1b Quotes

HW1a is about writing a typical, RFC based client side application. Furthermore, it sent passwords in the clear. HW1b is just the opposite. HW1b will involve both the client and server sides of a simple protocol, and will not send passwords in the clear. Both parts are due at 2:00 am Fri the 27th. We'll have separate instructions on how to do the submit.

For simplicity, both the client and server code will be implemented in `quote.pl` and a command line argument will determine which role the program is taking. For the first milestone, we'll consider the program without any passwords. The server blocks on a port as usual. The client connects, sends a "target" string, and the server sends back quotes that include that string and closes the connection.

Perl

In order to use the MD5 Perl library the top of your Perl file needs to look like this (the starter file is set up for you this way)...

```
#!/usr/pubsw/bin/perl5.6.1 -I/usr/class/cs193i/pm -w

use Socket;
use FileHandle;

use Digest::MD5 qw(md5 md5_hex md5_base64);
```

The first line says to use the perl5.6.1 version on leland (it works on the elaines; I have not tried the other machines). This is required because the older `/usr/bin/perl` is not compatible with the `Digest::MD5` module. The `-I` directive points to the `Digest::MD5` module which is in the "pm" directory in the 193i directory. You must run your perl program as "quote.pl" or "./quote.pl", so the "#!..." can specify which perl to use. The `Digest::MD5` module is available as open source from www.cpan.org if you'd like to make your own install, but if you just run on the elaines, we have it all installed already.

If you are having problems getting the MD5 module to work, it's possible to do the assignment without it, but it's not as neat.

Client Side

If the first command line argument is "-c" then the program should run in client mode. In client mode, the three other arguments are the name of the machine to connect to, the port to use, and (optional) the target string to use. If the target is not present, the client should treat it as the empty string. The client should connect to the server, send the target string on a line (the target may be the empty string), and read back and print out all the lines the server sends back. The quote protocol will use a simple "\n" as the line ending convention in both directions.

```
% ./quote.pl -c localhost 61784 "fish"
Even a fish could stay out of trouble if it would just learn to keep its mouth shut.
% ./quote.pl -c localhost 61784 "gas"
We don't have time to stop for gas -- we're already late.
% ./quote.pl -c localhost 61784
This time for sure!
%
```

In the above 'localhost' is a special DNS name that points to the local machine, and "./quote.pl" is a way to run a program in your current directory.

Server Side

If the first command line argument is "-s" then the program should run in server mode. In server mode, the program should take as input a port # to use and the name of a file with quotes on each line. Pick a random port number to use. The server should wait for incoming connections on that port. When the client connects, the server sends a "hello" greeting on a line by itself. The client ignores this greeting. The client sends the target on a line by itself. The server iterates through the quotes, sends back every quote that contains the target (each on its own line), and then closes the connection (we prefer case-insensitive search, but it is not required). If the target string is the empty string, then the server should select a quote at random and send that back.

```
% ./quote.pl -s 61784 quotes.txt
quote server 61784
connection from localhost
target 'fish'
connection from localhost
target 'gas'
connection from localhost
target ''
^C
```

The server should print out the initial "Quote server *port*" notice and then the two status messages for each connection. The server should read the quotes file into a global array when it starts up so the necessary data is all in memory when the connection comes in.

Error Handling

As with part (a), for error conditions, the program should print a standard, terse message to standard output and then `exit(-1)`. Please use the following error statements...

```
error hostname
error socket
error connect
error sockopt
error bind
error listen
error accept
```

Mechanics

The easiest way to test the program is with two terminal windows logged in to two machines, but in the same directory. Start the server in one terminal. Use `ctrl-c` to kill it when necessary. Run the client in the other terminal. When you edit `quote.pl`, you still need to kill and re-start the server to get it to use the new code. Use the "up-arrow" feature of the shell to retrieve a previous command without re-typing it.

Put lots of print statements throughout the client and server code so you can see what is going on. You can comment them out later.

Pitfall: after the `accept()` call in the server, set the newly created `CLIENT` socket to be unbuffered (the server socket handout does not show this step, but in this case you need it). If the client never gets things the server sends, buffering is the likely problem.

Milestone

Get the basic no-password version working reliably before trying the next step.

Authentication

We can't let just anyone look through our valuable quote database! We'll need to do some authentication to make sure that each party is who they say they are. The scheme below will allow the client and the server to each authenticate that the other knows what the password is. This is a little stronger than what POP does, where the client must authenticate, but the server does not. This scheme has some theoretical weaknesses vs. a determined bad guy, but it works well enough and it's a nice demonstration of 2-way challenge-response authentication that avoids sending the password over the network.

The server will control whether authentication is used. We'll add a `-p password` command line argument that follows the `-s`. If `-p` is present, the server will insist on authentication. In that case, the server sends the string "auth" as its initial greeting instead of "hello". The client can have a `-p password` command line argument after the `-c`. The client will use that password if the server insists on authentication. If the `-p` command is not present, the password defaults to the empty string.

1. The server sends "auth" as its greeting line, followed by `T` and `R1`, each on a line by itself. `T` is the current GMT time as a string. In Perl, use the function `gmtime()` to get this string. `R1` is a 4 digit random number (use `substr(rand(), 2, 4)`). These two pieces of information are the "challenge" to the client.
2. The client responds to the server challenge. The client makes a two line response: `hash(T,R1,P)`, where "hash" is the one way function described below, called on the `T`, `R1`, `P` strings. `P` is the password, which is known to the client. The second line of the client response is `R2`, a random number computed by the client.

3. The server responds to the client challenge by computing and sending hash(T, R2, P) on a line.
4. After exchanging challenge/response pairs, the client and server each print a "had:local-hash got:remote-hash" to local output comparing the hash expected vs. the hash received.
5. Now, the client and server can each decide if they are "happy" about the response they got -- did it match the hash expected. If the server is not happy, it prints "error: bad client password", sends "-quit" as the first quotation, and closes the connection. If the client is not happy, it prints "error: bad server password", sends "-quit" as the target string, and exits. Each end should detect if the other is sending "-quit", and close the connection gracefully and without printing the "-quit". Usually, both will be unhappy at the same time, but it's possible for one to be happy and one not if one is a "bad guy" program.
6. If both are happy, the quote protocol proceeds as it did without authentication. The client sends a target string, the server responds with quotes.

The hash function takes three strings and combines them into a token. The hash() function is present in the starter file. For initial testing, the hash function should just concatenate the three strings, separated by "*" characters as you can see in the have:/got: lines below. This is a terrible hash function, but it works and it's perfect for testing.

```
Server:
% ./quote.pl -s -p foo 61784 quotes.txt
quote server 61784
connection from localhost
have:Sat Apr 21 16:24:16 2001*3329*foo got:Sat Apr 21 16:24:16 2001*3329*foo
target ''
connection from localhost
have:Sat Apr 21 16:24:33 2001*8635*foo got:Sat Apr 21 16:24:33 2001*8635*foo
target 'gas'
connection from localhost
have:Sat Apr 21 16:24:52 2001*3586*foo got:Sat Apr 21 16:24:52 2001*3586*bar
error: bad client password
^C
```

```
Client:
% ./quote.pl -c -p foo localhost 61784
have:Sat Apr 21 16:24:16 2001*4301*foo got:Sat Apr 21 16:24:16 2001*4301*foo
Beware the lollipop of mediocrity -- lick it once and you suck forever.
% ./quote.pl -c -p foo localhost 61784 "gas"
have:Sat Apr 21 16:24:33 2001*2293*foo got:Sat Apr 21 16:24:33 2001*2293*foo
We don't have time to stop for gas -- we're already late.
% ./quote.pl -c -p bar localhost 61784 "gas"
have:Sat Apr 21 16:24:52 2001*8312*bar got:Sat Apr 21 16:24:52 2001*8312*foo
error:bad server password
%
```

Once the '*' version is working, set the constant `hash_md5` to 1. This changes the `hash()` function to use the real MD5 hash "one way" function -- it hashes the strings in a way that is believed to be very difficult to invert. That is, if a bad guy sees the hash, they will not be able to compute the (T, R, P) that made it (or at least it will be very computationally expensive to do so). Here's what the interaction looks like with real hashing (also, in this case I was running on two different machines, so the host names are more interesting)...

Sever:

```
elaine9:~/193i/quote> quote.pl -s -p foo 3465 quotes.txt
quote server 3465
connection from elaine0.Stanford.EDU
have:1Bdlhad4P565C/yzCqvjvg got:1Bdlhad4P565C/yzCqvjvg
target ''
connection from elaine0.Stanford.EDU
have:tYg5JA1XitX4vnCPbCKQjQ got:tYg5JA1XitX4vnCPbCKQjQ
target 'fish'
^C
```

Client:

```
elaine0:~/193i/quote> quote.pl -c -p foo elaine9 3465
have:XUmeAH9/wD9JilH4gXJ3DQ got:XUmeAH9/wD9JilH4gXJ3DQ
Even a fish could stay out of trouble if it would just learn to keep its mouth shut.
elaine0:~/193i/quote> quote.pl -c -p foo elaine9 3465 fish
have:xLBAR2HAuJldmonrS7pzLg got:xLBAR2HAuJldmonrS7pzLg
Even a fish could stay out of trouble if it would just learn to keep its mouth shut.
```

Please switch back to the "*" hash version for what you hand in -- the md5 version is harder to deal with for debugging and grading.