Feeding the Hungry Cookie Monster

Lab 4, CS 213, Fall, 1998

Assigned: Nov. 17, Due: Dec. 4, 11:59 PM

1 Logistics

- Larry Greenfield (leg+cs213@andrew) is the primary point of contact for questions.
- As always, you may work in a group of up to 2 people.
- All files and programs you will need are in CLASSDIR/L4/, where CLASSDIR is /afs/cs.cmu.edu/academic/class/15213-f98/.
- Any clarifications and revisions to the assignment will be posted on the class bboard and Web page.
- Problem 1 of the assignment must be done on our private ethernet, which is attached to interface *pf1* on the following Alpha systems: *mauve.ece*, *orange.ece*, *periwinkle.ece*, *purple.ece*, *slate.ece*, *umber.ece*, and *wheat.ece.*
- Since we'll be using *maroon.ece* to broadcast to the private ethernet, please do not use *maroon.ece* for any part of the assignment.

2 Introduction

A cookie monster is living somewhere on the CMU internet, patiently waiting to receive a cookie from each group in the class. Your job is to figure out where the cookie monster lives and what cookie he wants from your group (Problem 1), to send him his favorite cookie (Problem 2), and finally to write your own cookie monster (Problem 3).

3 Problem 1: Finding the cookie monster and his favorite cookie

The location of the cookie monster and the cookie he wants from each group is contained in a *cookie message* that is being injected continuously onto our private ethernet. The cookie monster's location is given as an IP host name and a TCP port number. Each cookie message is a sequence of ethernet packets, each of which encapsulates a *cookienet* packet whose format was devised by your instructors (with advice from the cookie monster of course). The cookie message will contain the cookie monster's location, and the specific cookie it is expecting from your group.

Your task in Problem 1 is (a) to write a packet monitor based on the Unix *packetfilter* interface, (b) to use your packet monitor to reverse-engineer the cookienet protocol, and (c) to use your results from (a) and (b) to decode the cookie message.

Starting the broadcast

We won't start broadcasting the cookie monster's location and the cookie he wants from your group until you send mail to Larry Greenfield (leg+cs213@andrew.cmu.edu) with the string *"Lab 4 cookie request"* in the subject header and the names and Andrew ID's of the group members in the message body. Larry will tell you when the broadcast has started and your group's number.

Note that this is a service provided by a real person, so don't expect an immediate response.

Writing a packet monitor

Your instructors have provided you with a framework in *pfr.c* that parses the command line arguments, finds and opens an available packetfilter, binds it to the private network, and then returns a file descriptor to that packetfilter that you can use in subsequent `read(2)` command to grab packets. After it acquires a file descriptor for an open packetfilter, *pfr* makes repeated calls to a pair of empty *readpacket()* and *writepacket()* routines.

The packetfilter provides a raw interface to the Ethernet layer and allows us to examine packets at the lowest level. Look at the `packetfilter(7)` man page for more information.

Your task in this section is to implement the *readpacket()* and *writepacket()* routines. *Readpacket()* should read packets until it encounters a cookienet packet (type `0x0011` in the ethernet header), and then return a pointer to that packet. *Printpacket()* should print the entire packet in hex format.

An ethernet header has the following format:

```
/* ethernet header */
unsigned char ether_dhost[6]; /* ethernet dst */
unsigned char ether_shost[6]; /* ethernet src */
unsigned short ether_type;    /* packet type */
                              /* cookienet packets are type 0x0011 */
```

You can test your initial version of the packet monitor by running your own instance of the *pfw* program, which is the same program that is sending out the cookie messages:

```
pfw [-h] [-f] [-r<n>] <port> < <ASCII file>
```

Pfw reads an ASCII file from the standard input and writes it to the private network as a sequence of one or more messages. Each message consists of the contents of the file written as a permuted sequence of rot(n) encrypted cookienet packets targeted to cookienet port `<port>`. The `<port>` argument is required and is used to distinguish your test messages from your cookie message and other students' test messages. The remaining arguments are optional:

-h Prints a help message.

-f Prints a continuous sequence of messages forever. Defaults to sending one message.

-r*n* Sets the rotation distance n for the rot(n) encryption (described in the next section). Defaults to $n = 0$ (i.e, -r0), which is no encryption.

Since at this point in the assignment you don't yet know the cookienet packet format, you will have to ignore the port number for now. You must use the contents of your test message to distinguish your packets from the cookie message and the other students' test messages.

Reverse-engineering the cookienet protocol

An encapsulated cookienet packet consists of a *header* and a *payload*. The payload is a sequence of rot(n) encrypted ASCII characters. A rot(n) encryption replaces each English letter with the letter n places forward along the alphabet. The parameter n is called the *rotation distance*. If c is the plaintext character, d the encrypted character, and n the rotation distance, then

$$d \leftarrow' a' + (((c -' a') + n)\%26)$$

if c is a lower case letter,

$$d \leftarrow' A' + (((c -' A') + n)\%26)$$

if c is an upper case letter, and

$$d \leftarrow c$$

otherwise. For example, y becomes 'z' in a rot(1) encryption and 'a' in a rot(2) encryption. Similarly, 'Y' becomes 'Z' in a rot(1) encryption and 'A' in a rot(2) encryption. Of course rot(0) corresponds to no encryption.

The header contains four fields (not necessarily in this order):

- The rot(n) rotation distance.
- A cookienet port number that disambiguates message sequences sent by different instances of *pfw*. The cookie message for your group is sent to the same port number as your group number. For exmaple, group 14 gets their cookie message from port 14.
- A sequence number that gives the order of the packet within the message.
- A size proportional to the length of the packet. The end of a message is indicated by a cookienet packet with a size field of zero.

Each field in the cookienet header (as well as the packet type in the ethernet header) is stored in network byte order (i.e., big-endian).

Your task in this section is use *pfw* and *pfr* to reverse-engineer the cookienet packet format.

Decoding the cookie message

Once you have reverse-engineered the cookienet protocol, you should write the final versions of *readpacket()* and *printpacket()* and then use this to decode the cookie message which is being continuously sent to the cookienet port with your group number.

Readpacket() should accept only cookienet packets that match the port number on the command line. *Printpacket()* should then print each packet using the following format:

```
printf("%04d:%s", seqno, payload);
```

where *seqno* is the cookienet sequence number and *payload* is a pointer to the plaintext version of the cookienet payload.

Important: Since we are doing automatic grading, you must output the packets in this format to get full credit. Also, you should not change the interface to the readpacket() and printpacket() routines.

4 Problem 2: Sending the cookie monster his cookie

You task in this problem is to write a sockets-based client program (`client.c`) that sends the cookie monster the cookie he wants from your group and waits for a reply. The cookie message will have details about format of the message to send to the cookie monster. If you send him the right cookie, he will reply with the message string "Yummy". If you send him the wrong cookie he will happily eat it, but he'll point out that it isn't his favorite cookie. Once you get a "Yummy" reply you are finished. Like the bomb, we will have a web page showing each groups current status. Unlike the bomb, you won't lose points for sending the wrong cookie.

The cookienet message you intercept in Problem 1 will describe the cookie protocol in detail—be sure to read it carefully. In brief, your client program should take four arguments: the host (either by name or IP address), the port, a group number, and a cookie to feed to the cookie monster:

```
% client cookie.monster.net 3212 14 "chocolate chip cookie"
yummy
%
```

Your client program should either print "yummy" if the cookie transfer was successful, "yucky" if the transfer was successful but the cookie monster didn't say "yummy", or "error:" followed by a descriptive error message.

Your client program should be robust: if an error occurs during cookie transmission, it should report it to the user.

Your client program should be written with the standard UNIX socket calls. Here are some suggestions for bvooks:

- *Using C on the UNIX System* by David A. Curry. This is a very short O'Reilly book.
- *UNIX Network Programming* by W. Richard Stevens. This is the classic reference and very comprehensive.
- *Internetworking with TCP/IP, volume III* by Douglas E. Comer and David L. Stevens. This is a very comprehensive book with a lot of examples (including many examples of concurrent servers).

The following UNIX man pages may be useful: `gethostbyname`, `socket`, and `connect`.

In addition, there are many small UNIX programs that open sockets. The following programs are available on Andrew:

- `imtest` (`/afs/andrew/system/src/local/cyrus/028/imtest/imtest.c`) is a program that opens a test connection to an IMAP (Internet Message Access Protocol) server. It was developed at CMU as part of the Cyrus project. (Try `imtest -kp cyrus.andrew.cmu.edu imap`.)

- `finger` (`/afs/andrew/system/src/local/fnger/030/finger/finger.c`) examines information about other users. It will open a connection to a remote host when the user isn't local (try `finger greenfie@math.rutgers.edu`).
- `whois` (`/afs/andrew/system/src/local/whois/013/whois.c`) queries an Internet database for information about various domains (try `whois cmu.edu`).

It can be very useful to use the `telnet` program to connect directly to the cookiemonster, to see how the server reacts directly to your queries. You can feed the cookiemonster this way, but you will not receive any credit for this part unless you turn in a working client program. To use `telnet` to connect to a specific port, try:

```
% telnet <host> <port>
```

5 Problem 3: Writing your own cookie monster

For this problem, you implement a concurrent cookie monster server of your own. You must conform strictly to the protocol given to you in Problem 1. A concurrent server must handle multiple simultaneous connections, which is normally done with *fork()*.

Since we're not attempting to test your ability to write a parser, you may use (at your option) the parser given in `parser.c`.

Your server should take three arguments: the first specifying what port your server will accept connections on, the second a group number, and the third the cookie monster's preferred cookie for that group.

The server's output (to standard out) should be of the form:

```
% server 3110 11 "chocolate chip cookie"
connection 1: connected
connection 1: hello group 4
connection 2: connected
connection 1: cookie "vanilla cookie": yucky
connection 2: hello group 11
connection 1: closed
connection 2: cookie "chocolate chip cookie": yummy
connection 2: closed
```

In addition to the reference books above, you may also find the man pages for `bind`, `accept`, and `fork` helpful.

Unfortunately, server programs are generally more complicated than client programs. Some real world examples are:

- `sendmail` (`/afs/andrew/system/src/local/sendmail/001/src/daemon.c`) is the major implementation of SMTP (Simple Mail Transfer Protocol), which forms the basis of internet mail. (Try `telnet smtp.andrew.cmu.edu smtp`.)
- `acapd` (`/afs/andrew/system/src/local/acapd/007/sockethandler.cc`) is a CMU implementation of ACAP (Application Configuration Access Protocol), which was developed as part of the Cyrus system. It is written in C++. (Try `telnet acap.andrew.cmu.edu acap`.)

6 Hand In

Before handing in, modify the `GROUP` file.

Modify your `Makefile` to correctly compile your client and server (the executables should be named `client` and `server`). You will be expected to handin `pfr.c`, `client.c`, and `server.c`. Make sure they compile with no additional files besides the included `.h` files.

Hand in your files with the command:

```
make handin NAME=yourname
```

where `yourname` is replaced with the Andrew ID of the first person on your team. Your solution will be copied to the handin directory. If you need to submit another version to fix a bug, use the command

```
make handin NAME=yourname VERSION=versionnumber
```

where `versionnumber` starts at 2 and counts up with each submission *after* the initial handin.