CS 422 (Park) **Assignment V** **Due: Apr. 13 (Wed.), 2005**

Submission instructions: Please type your answers and submit electronic copies using `turnin` *by 11:59pm on the due date. You may use any number of word processing software (e.g., Framemaker, Word, LaTeX), but the final output must be in pdf or ps format that uses standard fonts (a practical test is to check if the pdf/ps file prints on a CS Department printer without problem). For experiments and programming assignments that involve output to terminal, please use* `script` *to record the output and submit the output file. Use* `gnuplot` *to plot graphs. Use* `ps2gif` *to convert a eps/ps plot to gif format (e.g., for inclusion in Word).*

PROBLEM 1

Read Chapters 13 (Sections 13.1–13.12), 16, 18, 20, 21 and 24 from Comer.

PROBLEM 2 (40 = 30 + 10 pts)

(a) As a continuation of Problem 3, Assignment IV, clean up your server code so that it is ready for prime-time deployment. The server binary, `remote_commander`, will be executed as a daemon with a single command-line argument that specifies the port number on which the server should wait for client requests. A TCP client request has the simplified format

$$\#\ \textit{command}\ \#\ \textit{argument-1}\ \#\ \textit{argument-2}\ \#\ \cdots\ \#\ \textit{argument-n}\ \#$$

The server forks a child, hands off the task to its child, then returns to waiting for the next request. All other superfluous items we have added (e.g., sleep) for pedagogical purposes should be eliminated. The parsing part should be strengthened, however, so that malformed requests (e.g., request does not start with "#") do not crash the server. Instead the server sends back a message "error: malformed request" and returns to waiting for the next request. Test the server application by running 10 clients (modified to send requests following the simplified format) on 10 separate machines and logging the interaction using `script`. Pick 7 commands, with/without arguments, we have not used yet in our benchmarks. Pick 3 commands that are differently malformed.

(b) Technically speaking, what distinguishes a daemon from a regular server application? What are the programming steps needed to turn `remote_commander` into a bona fide daemon?

PROBLEM 3 (90 = 30 + 30 + 30 pts)

(a) As a continuation of Problem 2(a), what vulnerabilities does your server code have? What client requests can still crash (i.e., kill or stop) `remote_commander`? Note: sending a non-existent command should neither crash the server nor its child that executes the bogus command. Does sending an extremely long request crash the server? Find at least 2 different means to crash `remote_commander`. Demonstrate that they indeed work. If your server code is lousily written to begin, they won't count. For each additional means of crashing `remote_commander`, you will get 10 bonus points (up to 30 bonus points).

(b) Can you find ways to effect denial-of-service (DoS) on your server? Crashing is an extreme form of DoS where other clients are hindered from accessing the service but, in general, a "DoS attack" need not crash a server to affect denial-of-service to others. Find at least 2 different DoS attacks and demonstrate how they work by running multiple clients with one instituting the attack. Can your server code be modified to withstand the DoS attacks? If so, how?

(c) Can you find a way for the client, by sending a request, to hijack the server process without in the process crashing it? Hijacking means that the attacker takes control of the server process by running parasite code transmitted as part of the request. First, describe logically an approach that may achieve the goal and the technical challenges contained therein. Then try to get it to work. If you succeed, you get 100 bonus points. How should your server code be modified to defend against this vulnerability (this is more subtle than it seems)?

PROBLEM 4 (50 pts)

Write a UDP-based traffic flooding application—sender and receiver—where the sender, `udp_flood`, takes two command-line arguments

```
% udp_flood dest-IP dest-port payload-size packet-count
```

where `dest-IP` is the destination IP address, `dest-port` the destination port, `payload-size` is the UDP payload size (in bytes), and `packet-count` is the total number of packets it is supposed to send before terminating. `udp_flood` follows a `for`-loop where it uses `sendto`() to transmit `packet-count` packets back-to-back without pause. The sender inscribes a sequence number, starting with 0, in the first 4 bytes of the payload. The receiver, `udp_receive`, takes two command-line arguments

```
% udp_receive local-port log-file
```

where `local-port` is the port on which it waits for packets and `log-file` is the name of a file into which it dumps measurement logs before terminating. `udp_receive` is essentially an infinite loop with a single `recvfrom`() blocking call that upon return logs the sequence number (4-byte unsigned integer inscribed by the sender in the first 4 bytes of the payload) into main memory (a pre-allocated array). It is important not to write to disk during the run as disk I/O will cause significant slow-down of the receiver application leading to receiver buffer overrun. The receiver application is terminated by entering `CNTL-C` using the keyboard which sends the SIGINT signal to the running foreground process. Since the default disposition of processes to SIGINT is to terminate, register a SIGINT signal handler that catches the signal when it is raised, writes the log data in memory to disk, and only then exits gracefully. Use `gnuplot` to generate three plots: (1) an impulse plot showing the packets received (the x-axis is the sequence number and the y-axis shows an unit impulse, i.e., vertical line, if the packet with that sequence number was received), (2) an impulse plot showing the packets not received (the opposite of (1)), and (3) a throughput plot at 1 second aggregation (the x-axis shows time in unit of 1 second and the y-axis shows the total bits received during a 1 second time interval; you need to consider the payload size, UDP header size, and Ethernet header/trailer size when computing the total bit of a packet). Benchmark your UDP flooding application using `payload-size` 128 bytes and `packet-count` 1,000,000 packets. Repeat the experiment by increasing `payload-size` to 512 bytes and 1024 bytes, respectively. Along with the 9 plots, write a 1-page report discussing your results. You do not need to submit the raw log files but you must have them ready to supply to the TAs when so requested.