Advanced Unix System Administration
Spring 2007
Homework 3-4 Solutions

1. *Networking on paper.* Here's an exercise to test your understanding of TCP/IP over Ethernet.

   a. Your company needs five different networks, four small networks of about 20 servers each, and a larger network of clients with addresses assigned by DHCP. You have the IP address range `172.17.42.0/24` to work with (I know this is RFC 1918 space – it's an example). Suggest a way to divide up this netblock into the networks you need.

      One solution: use `172.17.42.0/27`, `172.17.42.32/27`, `172.17.42.64/27`, and `172.17.42.96/27` for the servers and `172.17.42.128/25` for the clients. Each of the server networks will have 30 usable addresses (27 bits for the network address leaves 5 bits for the host address, which is 32 possibilities; the low value is used to identify the network, whereas the high value is the broadcast address), whereas the client network will have 126 addresses.

   b. A computer on an Ethernet network with MAC address `FF:FF:FE:09:42:A3` and IP address `172.17.42.37` sends the message `Hello, world!\r\n` via UDP from port 51500 to a computer with MAC address `FF:FF:FB:3D:28:9C` and IP address `172.17.42.58` on port 9. Write out a transcript of the Ethernet frames resulting from this conversation. Assume the sender's ARP cache is empty at the beginning of the conversation.

      This is a short conversation – three packets. First, the ARP broadcast looking for `172.17.42.58`:

```
Ethernet II frame
Destination: FF:FF:FF:FF:FF:FF (broadcast)
Source: FF:FF:FE:09:42:A3
Type: 0x0806 (ARP)
ARP packet
Hardware type: 0x0001 (Ethernet)
Protocol: 0x0800 (IP)
Opcode: 0x0001 (request)
Sender MAC: FF:FF:FE:09:42:A3
Sender IP: 172.17.42.37
Target MAC: 00:00:00:00:00:00
Target IP: 172.17.42.58
```

      The reply to the ARP request is not broadcast:

```
Ethernet II frame
```

```
Destination: FF:FF:FE:09:42:A3
Source: FF:FF:FB:3D:28:9C
Type: 0x0806 (ARP)
ARP packet
Hardware type: 0x0001 (Ethernet)
Protocol: 0x0800 (IP)
Opcode: 0x0002 (reply)
Sender MAC: FF:FF:FB:3D:28:9C
Sender IP: 172.17.42.58
Target MAC: FF:FF:FE:09:42:A3
Target IP: 172.17.42.37
```

With the MAC address of the destination in hand, the message is sent as a single UDP packet:

```
Ethernet II frame
Destination: FF:FF:FB:3D:28:9C
Source: FF:FF:FE:09:42:A3
Type: 0x0800 (IP)
IPv4 packet
Version: 4
Header length: 20 bytes
DSCP: 0x00
ECN: 0x00
Total length: 43 bytes
IPID: 32181 [could be anything]
Flags: 0x04 (DF bit set, MF bit clear)
Fragment offset: 0
TTL: 64 [could be more or less, depending on the IP stack]
Protocol: 0x11 (UDP)
Source: 172.17.42.37
Destination: 172.17.42.58
UDP packet
Source port: 51500
Destination port: 9
Length: 23 bytes
Data: Hello, world!\r\n
```

c. A computer on an Ethernet network with MAC address `FF:FF:FE:09:42:A3` and IP address `172.17.42.37` initiates a TCP connection from port 51501 to a computer with MAC address `FF:FF:FB:3D:28:9C` and IP address `172.17.42.58` on port 7. The computer on `.37` sends the string `Hello, world!\r\n` to the peer, which echos back the same message; the two computers then close the connection. Write out a transcript of the Ethernet frames resulting from this

conversation. Assume the initiating host's ARP cache already contains the entry for the machine it wishes to talk to.

Because of the way the TCP teardown can work, there are a few different possibilities for what exactly happens during this conversation; here's one. Note that I haven't used any TCP options here, which keeps the packets simpler; a real conversation is likely to use at least selective acknowledgment and TCP window scaling.

```
Ethernet II frame
Destination: FF:FF:FB:3D:28:9C
Source: FF:FF:FE:09:42:A3
Type: 0x0800 (IP)
IPv4 packet
Version: 4
Header length: 20 bytes
DSCP: 0x00
ECN: 0x00
Total length: 60 bytes
IPID: 51710 [could be anything]
Flags: 0x04 (DF bit set, MF bit clear)
Fragment offset: 0
TTL: 64 [could be more or less, depending on the IP stack]
Protocol: 0x06 (TCP)
Source: 172.17.42.37
Destination: 172.17.42.58
TCP packet
Source port: 51501
Destination port: 7
Sequence number: 1000 [subject to requirements on ISNs]
Acknowledgment number: 0
Header length: 20 bytes
Flags: 0x02 (SYN)
Window size: 5840 [depends on TCP stack and link]
```

```
Ethernet II frame
Destination: FF:FF:FE:09:42:A3
Source: FF:FF:FB:3D:28:9C
Type: 0x0800 (IP)
IPv4 packet
Version: 4
Header length: 20 bytes
DSCP: 0x00
ECN: 0x00
```

```
Total length: 60 bytes
IPID: 0 [could be anything]
Flags: 0x04 (DF bit set, MF bit clear)
Fragment offset: 0
TTL: 64 [could be more or less, depending on the IP stack]
Protocol: 0x06 (TCP)
Source: 172.17.42.58
Destination: 172.17.42.37
TCP packet
Source port: 7
Destination port: 51501
Sequence number: 2000 [subject to requirements on ISNs]
Acknowledgment number: 1001
Header length: 20 bytes
Flags: 0x12 (ACK|SYN)
Window size: 5792 [depends on TCP stack and link]
```

From this point, we only describe source and destination IP addresses, sequence and acknowledgment numbers, TCP flags, window sizes, and data for each packet.

```
Source: 172.17.42.37
Destination: 172.17.42.58
Sequence number: 1001
Acknowledgment number: 2001
Flags: 0x10 (ACK)
Window size: 5856
```

At this point, the three-way handshake is complete, and a TCP connection has been established. Notice the way the window size has grown – no packets have been dropped, so TCP allows more data to be transferred before the next ACK.

```
Source: 172.17.42.37
Destination: 172.17.42.58
Sequence number: 1001
Acknowledgment number: 2001
Flags: 0x18 (ACK|PSH)
Window size: 5856
Data: Hello, world!\r\n
```

Note that the sequence number for the ACK is reused. Sequence numbers are assigned for each byte of data, since it is data transmissions that need to be acknowledged; empty ACK packets need not be acknowledged, so they do not

need to take up sequence number space. (I incorrectly stated in class that sequence numbers are assigned per packet; this has now been corrected in the slides for March 7.) The PSH flag is set to tell the remote TCP stack to flush its buffers to the application, since this is the logical end of a transmission; if the transmission encompassed multiple packets, PSH would only be set on the last of these.

```
Source: 172.17.42.58
Destination: 172.17.42.37
Sequence number: 2001
Acknowledgment number: 1016
Flags: 0x10 (ACK)
Window size: 5824
```

Notice the ACK number has jumped to 1016; sequence numbers are assigned per byte of data, and our data was 15 bytes long. Once the conversation starts, the empty ACK packet is strictly not necessary, as the packet can be acknowledged by the next data packet, but most TCP stacks emit these.

```
Source: 172.17.42.58
Destination: 172.17.42.37
Sequence number: 2001
Acknowledgment number: 1016
Flags: 0x18 (ACK|PSH)
Window size: 5824
Data: Hello, world!\r\n
```

The server echos data back to the client.

```
Source: 172.17.42.37
Destination: 172.17.42.58
Sequence number: 1016
Acknowledgment number: 2016
Flags: 0x10 (ACK)
Window size: 5856
```

```
Source: 172.17.42.37
Destination: 172.17.42.58
Sequence number: 1016
Acknowledgment number: 2016
Flags: 0x11 (ACK|FIN)
Window size: 5856
```

The client is telling the server that it has no more data to send. Note that the server can continue to send data to the client; this is known as a "half-open" connection.

```
Source: 172.17.42.58
Destination: 172.17.42.37
Sequence number: 2016
Acknowledgment number: 1017
Flags: 0x11 (ACK|FIN)
Window size: 5824
```

FIN packets need to be acknowledged to complete the tear-down of the connection, so the ACK number has increased despite the fact that no further data has been transmitted. The FIN flag is set to indicate that the server also has no more data to send; again, a FIN from one side is not necessarily followed immediately by a FIN from the other.

```
Source: 172.17.42.37
Destination: 172.17.42.58
Sequence number: 1017
Acknowledgment number: 2017
Flags: 0x10 (ACK)
Window size: 5856
```

This last ACK from the client completes the tear-down of the TCP connection in both directions.

Notice the overhead from the TCP connection; what would have taken two packets to say in UDP has taken 10 packets to say in TCP. TCP is therefore frequently undesirable for conversations like this, consisting of short, discrete messages; on the other hand, if transmitting larger streams of information, or if reliability is an issue, the overhead is less of a problem.

2. *Recursive DNS lookups.* Resolve the following DNS queries by hand using `dig +norecurse`:

   - `mail.Math.Berkeley.EDU IN A`
   - `www.google.com IN A`
   - `bigsur.steven676.net IN MX`.

   By "by hand", I mean you get to do the recursion yourself, starting from the root DNS servers. The `-t` option allows you to specify a record type, and `@` is used to specify a target DNS server. Include each `dig` command you used and the output. Hint: one of the options to `dig` allows you to check your work quite easily.

   The general idea here is that we query DNS servers recursively for what we want, starting at the root; either the server will have a reply for us (in which case we'll have one or more answers), or it will give us a referral to a server that will know more (no answers, but authorities in the response).

```
$ dig +norecurse @f.root-servers.net mail.Math.Berkeley.EDU

; <<>> DiG 9.3.4 <<>> +norecurse @f.root-servers.net mail.Math.Berkeley.EDU
; (1 server found)
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 55612
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 8, ADDITIONAL: 8

;; QUESTION SECTION:
;mail.Math.Berkeley.EDU.                IN      A

;; AUTHORITY SECTION:
EDU.                    172800  IN      NS      L3.NSTLD.COM.
EDU.                    172800  IN      NS      M3.NSTLD.COM.
EDU.                    172800  IN      NS      A3.NSTLD.COM.
EDU.                    172800  IN      NS      C3.NSTLD.COM.
EDU.                    172800  IN      NS      D3.NSTLD.COM.
EDU.                    172800  IN      NS      E3.NSTLD.COM.
EDU.                    172800  IN      NS      G3.NSTLD.COM.
EDU.                    172800  IN      NS      H3.NSTLD.COM.

;; ADDITIONAL SECTION:
A3.NSTLD.COM.           172800  IN      A       192.5.6.32
C3.NSTLD.COM.           172800  IN      A       192.26.92.32
D3.NSTLD.COM.           172800  IN      A       192.31.80.32
E3.NSTLD.COM.           172800  IN      A       192.12.94.32
G3.NSTLD.COM.           172800  IN      A       192.42.93.32
H3.NSTLD.COM.           172800  IN      A       192.54.112.32
L3.NSTLD.COM.           172800  IN      A       192.41.162.32
M3.NSTLD.COM.           172800  IN      A       192.55.83.32
```

We haven't gotten an answer, but we get a referral to the .edu DNS servers (apparently `*3.nstld.com`) instead. In the interests of brevity, I'll omit further `dig` output unless it's otherwise interesting.

```
$ dig +norecurse @L3.NSTLD.COM mail.Math.Berkeley.EDU
[snip]
$ dig +norecurse @ADNS1.Berkeley.EDU mail.Math.Berkeley.EDU

; <<>> DiG 9.3.4 <<>> +norecurse @ADNS1.Berkeley.EDU mail.Math.Berkeley.EDU
; (1 server found)
;; global options:  printcmd
```

```
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 46569
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 6, ADDITIONAL: 6

;; QUESTION SECTION:
;mail.Math.Berkeley.EDU.                IN      A

;; ANSWER SECTION:
mail.Math.Berkeley.EDU. 10800   IN      A       169.229.58.57

;; AUTHORITY SECTION:
Berkeley.EDU.           172800  IN      NS      phloem.uoregon.EDU.
Berkeley.EDU.           172800  IN      NS      ucb-ns.NYU.EDU.
Berkeley.EDU.           172800  IN      NS      adns1.Berkeley.EDU.
Berkeley.EDU.           172800  IN      NS      adns2.Berkeley.EDU.
Berkeley.EDU.           172800  IN      NS      dns2.ucla.EDU.
Berkeley.EDU.           172800  IN      NS      ns.v6.Berkeley.EDU.

;; ADDITIONAL SECTION:
ns.v6.Berkeley.EDU.     172800  IN      A       128.32.136.6
ns.v6.Berkeley.EDU.     172800  IN      AAAA
2001:468:e21:0:2a0:c9ff:fea0:110d
dns2.ucla.EDU.          21600   IN      A       164.67.128.2
adns1.Berkeley.EDU.     172800  IN      A       128.32.136.3
adns2.Berkeley.EDU.     172800  IN      A       128.32.136.14
phloem.uoregon.EDU.     86400   IN      A       128.223.32.35
```

That gets us our answer:

```
mail.Math.Berkeley.EDU. 10800   IN      A       169.229.58.57
```

Similarly, by doing the following, we find:

```
$ dig +norecurse @f.root-servers.net www.google.com
[snip]
$ dig +norecurse @C.GTLD-SERVERS.NET www.google.com
[snip]
$ dig +norecurse @ns1.google.com www.google.com

; <<>> DiG 9.3.4 <<>> +norecurse @ns1.google.com www.google.com
; (1 server found)
;; global options:  printcmd
;; Got answer:
```

```
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 37278
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 7, ADDITIONAL: 7

;; QUESTION SECTION:
;www.google.com.                        IN      A

;; ANSWER SECTION:
www.google.com.         604800  IN      CNAME   www.l.google.com.

;; AUTHORITY SECTION:
l.google.com.           86400   IN      NS      a.l.google.com.
l.google.com.           86400   IN      NS      b.l.google.com.
l.google.com.           86400   IN      NS      c.l.google.com.
l.google.com.           86400   IN      NS      d.l.google.com.
l.google.com.           86400   IN      NS      e.l.google.com.
l.google.com.           86400   IN      NS      f.l.google.com.
l.google.com.           86400   IN      NS      g.l.google.com.

;; ADDITIONAL SECTION:
a.l.google.com.         86400   IN      A       209.85.139.9
b.l.google.com.         86400   IN      A       64.233.179.9
c.l.google.com.         86400   IN      A       64.233.161.9
d.l.google.com.         86400   IN      A       66.249.93.9
e.l.google.com.         86400   IN      A       209.85.137.9
f.l.google.com.         86400   IN      A       72.14.235.9
g.l.google.com.         86400   IN      A       64.233.167.9
$ dig +norecurse @a.l.google.com www.l.google.com
[snip]
```

Notice we get a CNAME record as a reply; RFC 1034 then tells us we should restart the query at the domain name of the CNAME reply, with the query name changed to the one provided us in the CNAME reply. Hence we follow the query of `ns1.google.com` for `www.google.com` by a query of `a.l.google.com` for `www.l.google.com`. The reply returned by a nameserver performing this query will include both the CNAME record and the A records eventually returned, so the answer here is

```
www.google.com.         604800  IN      CNAME   www.l.google.com.
www.l.google.com.       300     IN      A       72.14.253.103
www.l.google.com.       300     IN      A       72.14.253.104
www.l.google.com.       300     IN      A       72.14.253.99
www.l.google.com.       300     IN      A       72.14.253.147
```

The multiple A records serve as a DNS round-robin.

We proceed as follows for the last record to be looked up:

```
$ dig +norecurse @f.root-servers.net -t MX bigsur.steven676.net
[snip]
$ dig +norecurse @H.GTLD-SERVERS.NET -t MX bigsur.steven676.net
[snip]
$ dig +norecurse @ns1.sitelutions.com -t MX bigsur.steven676.net

; <<>> DiG 9.3.4 <<>> +norecurse @ns1.sitelutions.com -t MX
bigsur.steven676.net
; (1 server found)
;; global options:  printcmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NXDOMAIN, id: 58458
;; flags: qr aa; QUERY: 1, ANSWER: 0, AUTHORITY: 1, ADDITIONAL: 0

;; QUESTION SECTION:
;bigsur.steven676.net.          IN      MX

;; AUTHORITY SECTION:
steven676.net.          3600    IN      SOA     ns1.sitelutions.com.
steven676.myrealbox.com. 75 28000 7200 604800 5
```

Notice the return status of `NXDOMAIN` here; the server is indicating that the requested record does not exist. The SOA record for the domain is returned as an authority answer.

The useful option referred to in the hint is the `+trace` option to `dig`, which performs a similar automated recursive lookup.

3. *Idle scan.* TCP initial sequence numbers aren't the only numbers that are problematic if they are predictable. There's an interesting technique called "idle scan", implemented in recent versions of `nmap`, that relies on a "zombie" host whose IPID numbers are predictable.

   a. How does this scan work? Why does the zombie host have to be idle? Where do the predictable IPID numbers come in?

      The scanning host first sends a packet to the zombie, and looks at the reply to collect its current IPID number. The scanning host sends a TCP SYN packet with source address forged to be that of the zombie host to the scan target. If the port is open on the scan target, the target will reply to the zombie with a SYN|ACK, which will cause the zombie to reply with an RST packet; this causes the IPID number of the zombie's packets to increase in the predictable manner previously detected. If the port is closed, an RST will be sent to the

zombie, which silently drops the packet; the zombie's IPID number does not increase in this scenario. Hence, by querying the zombie's IPID number before and after this sequence, we may determine whether or not the queried port was open.

This scheme breaks down if the zombie has any traffic other than the scan traffic, because increases in the IPID number will then not necessarily be correlated to whether the port on the target is open or not.

b. From where does the scan appear to be coming from, the scanning host or the zombie? Why? Why might this be a problem if a zombie on your network is being used to scan one of your machines?

The scan appears to be coming from the zombie, since this is the source address the scan target sees. This means that the scan results reflect the perspective of the zombie; if an attacker can find a zombie on your network, then, he can scan your machines as if he were inside your network, thus possibly evading some of your firewall rules.

c. What can you do to prevent idle scans from being launched from inside your network?

Since conducting an idle scan depends on the ability to send packets with spoofed source addresses, you can prevent idle scans of hosts outside your network from being launched inside your network by implementing egress filtering of packets with clearly spoofed source addresses (i.e. source addresses outside your network).

4. *Secure temporary file creation.* This is another one of those problems that turns out to be annoyingly hard, and comes back to bite us again and again.

a. Have a look at `tmpfile3.c`, which attempts to create a temporary file securely. Compile and run it, and watch its behavior (perhaps with `strace`). What's wrong with this approach? (Hint: this works fine on a single-tasking system.) *Optional:* Construct an exploit for this.

The trouble stems from this sequence:

```
/* Check to see if our chosen filename exists yet */
if (lstat(filename, &filestat) != -1 || errno != ENOENT) {
        /* Some error handling code */
}
/* Create the temporary file, permissions 0600 */
if ((fd = open(filename, O_WRONLY|O_CREAT, 0600)) == -1)
        return 1;
```

This translates into the following sequence of syscalls, under normal circumstances:

```
lstat("/tmp/tmpfile3.mRS0K5", 0x7fff4251fd90) = -1 ENOENT
    (No such file or directory)
open("/tmp/tmpfile3.mRS0K5", O_WRONLY|O_CREAT, 0600) = 3
```

There's nothing to guarantee that the situation stayed the same between the `lstat()` and the `open()`; in other words, we could very well have two processes A and B doing the following:

```
A: lstat("/tmp/tmpfile3.mRS0K5", 0x7fff4251fd90) = -1 ENOENT
    (No such file or directory)
B: symlink("/path/to/victim", "/tmp/tmpfile3.mRS0K5") = 0
A: open("/tmp/tmpfile3.mRS0K5", O_WRONLY|O_CREAT, 0600) = 3
```

A is now working on a file that it didn't intend to work with; this could allow an attacker to cause it to read arbitrary data of his choosing, or cause it to write to an arbitrary file. If `open()` is called with the `O_TRUNC` flag, the victim file is now empty, its data being clobbered; one could easily see this being a problem when A is running as root and the victim is, say, `/etc/passwd`.

This type of problem, where the exploit depends on fitting in an action between two or more other actions, is known as a race condition – the attacking process B is, in effect, "racing" process A to get the `symlink()` call in before A executes its `open()`. When the system isn't under load, the window of opportunity for exploit is often extremely short; thus attackers will frequently generate system load, hoping to slow down process A and thus make exploiting the bug easier.

b. Look at `tmpfile4.c`. What's changed from `tmpfile3.c`? Compile and run it, and watch its behavior with `strace`. Why does this work (on POSIX-compliant filesystems)? Is there something superfluous in what this program does?

   The big difference here is that `open` is now called with the `O_EXCL` flag:

```
lstat("/tmp/tmpfile4.1VJuus", 0x7fffc6c9d3a0) = -1 ENOENT
    (No such file or directory)
open("/tmp/tmpfile4.1VJuus", O_WRONLY|O_CREAT|O_EXCL, 0600) = 3
```

   POSIX guarantees that `open()` will fail to create a file if `O_EXCL` is specified and the filename already exists.

   We noted already that the `lstat()` is ineffective in ensuring that the file doesn't exist already; hence there's no real point in calling it. An `ltrace -S` of `tmpfile4` reveals that this `lstat()` is being performed by `mktemp()`.

c. You may have gotten a warning to the effect that `mktemp()` is dangerous, and that `mkstemp()` or `tmpfile()` should be used instead. In naive applications, such as `tmpfile3.c`, `mktemp()` is indeed dangerous, though `tmpfile4.c` demonstrates a correct use (though this could still be vulnerable if the implementation of `mktemp()` is bad, as on many historic Unixes). Watch the execution of `tmpfile5`, which is `tmpfile4` modified to use `mkstemp()` (as

modern practice recommends), with `strace`. What (if anything) does it do differently? Also watch the execution of the `mktemp` command, which (along with `tempfile`) is the recommended way of creating temporary files from shell scripts. Does it work any differently from `tmpfile4` and `tmpfile5`?

The interesting part of the `strace` output for `tmpfile5` is this:

```
open("/tmp/tmpfile5.FBOS9U", O_RDWR|O_CREAT|O_EXCL, 0600) = 3
```

Note that `mkstemp()` doesn't bother to do the superfluous `lstat()`. An `strace` of `mktemp` reveals that it does the same thing as `tmpfile5` – very possibly because it calls `mkstemp()` to create the temporary file.

d. On POSIX filesystems, the methods used by `tmpfile4` and `tmpfile5` are sufficient to guarantee that nothing bad will happen when a temporary file is opened. Why jump through hoops to obtain a unique, difficult-to-guess filename, in that case? Examine `tmpfile6.c`. For some applications, this is acceptable (indeed, it is occasionally necessary to create files in world-writable directories with predictable names, such as lock files); when might this be a problem, and why?

Choosing a predictable name causes a problem if the application insists on having the temporary file created with that name; an attacker would be able to cause a denial-of-service condition by preventing the application from creating its temporary file.

e. Over NFS (which isn't POSIX-compliant), the method of `tmpfile6.c` is insufficient to guard against a symlink attack. (`tmpfile4.c` and similar are also theoretically vulnerable, but as they choose difficult-to-guess temporary file names, conducting such an attack is hard; this is another good reason to avoid predictable temporary file names.) Look at `tmpfile7.c`, which creates a predictable file name in an NFS-safe manner. What's changed? Why does this work? *Note:* These issues mean that it's probably a good idea to avoid having `/tmp` and `/var/tmp` on NFS if at all possible; while applications creating lock files are generally aware of these issues, other users of `/tmp` may not be quite so clued in.

`tmpfile7` first creates a hopefully unique name, then creates a hard link to the desired name (which is predictable, since it's based on the program's PID) and removes the first name:

```
/* Select a temporary file name */
if (!(temp_filename = mktemp(temp_filename)))
        return 1;

/* Create the temporary file, permissions 0600 */
if ((fd = open(temp_filename, O_WRONLY|O_CREAT|O_EXCL, 0600)) == -1) {
        perror("File creation failed");
```

```
        return 1;
}

/* Create a hard link from our random filename to the desired one */
if (link(temp_filename, filename) == -1) {
        /* Weird NFS error handling logic */
}
/* Remove the random name, leaving us with just the desired one
 * Remember this doesn't affect the open file descriptor */
unlink(temp_filename);
```

This leads to the following sequence of syscalls, under normal circumstances:

```
lstat("/tmp/tmpfile7.q3Mbkk", 0x7fff6da7c0e0) = -1 ENOENT
    (No such file or directory)
open("/tmp/tmpfile7.q3Mbkk", O_WRONLY|O_CREAT|O_EXCL, 0600) = 3
link("/tmp/tmpfile7.q3Mbkk", "/tmp/tmpfile7.6424") = 0
unlink("/tmp/tmpfile7.q3Mbkk")          = 0
```

This is safe (or as safe as possible over NFS, anyway – it's possible that someone could guess our temporary file name between the `lstat()` and the `open()`) because `link()` is atomic and always fails if the target name already exists, even on NFS, thus giving an attacker no chance to interpose.

The "weird NFS error handling logic", by the way, is due to another oddity of the NFS protocol:

```
/* It's possible that link creation succeeds, but that the NFS
 * server crashes before it can return success to us; hence we
 * check the link count to see if it's increased */
if (fstat(fd, &filestat) == -1) {
        unlink(temp_filename);
        return 1;
}
if (filestat.st_nlink != 2) {
        fprintf(stderr, "Link creation failed\n");
        unlink(temp_filename);
        return 1;
}
```

5. *An exercise in setuid/setgid design – designing a secure local file sharing solution.* Suppose the users on your system want a way to copy files amongst themselves, except that (1) you want to be able to place restrictions on what can be copied between users and (2) your users want to be able to place restrictions on what can be copied to their home directories. Specifically, assume that these restrictions are

implemented as shell scripts (or other executables) that are run on each file to be copied.

a. Design a system using *a single binary* to securely perform this task (i.e. write down, in detail, the steps that such a system would take while copying a file from one user to another). Your security model may require the creation of new system users, if appropriate.

   There are many different ways of doing this; one possible method is to create a setuid root binary that does the following:

   i. Forks a child process, which uses `setuid()` to change to the calling user and calls the shell scripts implementing system-wide and per-user policies. The parent `wait()`s for its child to finish.
   ii. Checks the return status of its child to determine whether or not the file passed policy checks. If not, exit immediately.
   iii. Opens the file for reading.
   iv. Uses `setuid()` to change to the user receiving the file.
   v. Opens the output file for writing.
   vi. Copies the contents of the file to the appropriate location.

   Notice that policy checks are run as the calling user, limiting the usefulness of attempting to exploit the policy checking scripts; this does have the side effect of requiring that the scripts be readable by all users allowed to invoke the binary, though with some further cleverness this restriction can be evaded somewhat. The amount of time spent running as root is kept to an absolute minimum.

b. Design a system to perform this task as securely as possible. You may use as many binaries and/or running daemons as you think appropriate. Your security model may require the creation of new system users, if appropriate.

   Again, there are many different ways of doing this. One approach is to split up the binary above into two separate ones, a frontend that's setgid to a specific group, and a backend that's setuid root and only executable by that group. The delivery proceeds as follows:

   i. The frontend is invoked.
   ii. The frontend calls `setegid()` to temporarily set its effective GID to its real GID (the GID of the calling user).
   iii. The frontend forks a child process (which, since the effective GID is different, will not inherit the privileged group), then calls the shell scripts implementing system-wide and per-user policies. The parent `wait()`s for its child to finish.
   iv. The frontend checks the return status of its child to determine whether or not the file passed policy checks. If not, exit immediately.

v. The frontend calls `setegid()` to set its effective GID to the saved GID (the privileged GID).

vi. The frontend calls the backend.

vii. The backend opens the file for reading.

viii. The backend uses `setuid()` to change to the user receiving the file.

ix. The backend opens the output file for writing.

x. The backend copies the contents of the file to the appropriate location.

The elaborate dance that the frontend does is to ensure that the policy shell scripts are never invoked with the privileged group, which would allow them to possibly be exploited to call the backend to deliver the file. In this design, the amount of privilege granted, the amount of code running privileged, and the amount of time spent running privileged are all kept to a minimum.

Another possibility involves having the backend run as a daemon reading out of a queue which is only writable by the privileged group. The policy checks could also be implemented as daemons.

c. Which design is more secure? Why?

The multi-binary design reduces the amount of code that could potentially be run as root, and is therefore a more secure design.

6. *Setting up a `chroot()` jail.* While not foolproof, a `chroot()` jail can help improve the security of a service by making an attacker's life more difficult. Here, you get to set up chrooting for some programs; while they don't do much useful, they could potentially be run out of `inetd` in their given forms.

a. Log in to your container and have a look at `chroot1.c`. Compile and run (you'll need to be root to run, as the use of `chroot()` is restricted to root), and watch execution with `strace`. Why is the `chdir()` after the `chroot()` necessary? *Optional:* For bonus points, why do we need to drop privileges?

The `chdir()` is necessary because `chroot()` does not necessarily change the working directory of the process (and definitely doesn't on Linux and Solaris); hence, unless the process's current working directory is already under the target directory, it will end up outside the `chroot()` jail when `chroot()` is called. This makes it trivially easy to break out of the jail – repeated calls of `chdir("..")` will get us to the root of the real directory tree.

Dropping privileges is necessary because it is possible for a privileged program to break out of a `chroot()` jail. Consider the following sequence:

i. Program creates a directory `temp` under its current working directory.

ii. Program `chroot()`s into `temp` (this requires privileges). At this point, the current working directory is outside the process's root directory.

iii. Program repeatedly does `chdir("..")` to move up in the directory tree, eventually to the real root directory; it can do this, since its current working directory is outside the supposed root directory.

iv. Program calls `chroot(".")` to change its root to the real root directory.

(It's a bit more complicated if `chroot()` changes the current working directory, but not impossible.) By dropping privileges, we ensure that the process cannot use `chroot()` to do this.

b. `chroot2.c` omits the built-in chrooting of `chroot1.c`. Compile it, and set up an environment in `/chroot` in which you can run it with the `chroot` command. Hint: you'll need to copy some files into `/chroot`; `ldd` and `strace` should help you figure out which ones. In general, you want to keep a chroot as empty as possible, to limit the possibilities an attacker has inside.

Besides the program itself, you only need two files in your `chroot()` jail, the dynamic linker and the C library. On an AMD64 Linux box, that means `/lib64/ld-linux-x86-64.so.2` and `/lib/libc.so.6`.

c. `chroot3.c` and `chroot4.c` are identical, except that `chroot4.c` has built-in chrooting, while `chroot3.c` does not. Set up an environment in `/chroot` in which `chroot4` runs and produces identical output to `chroot3` run in the main filesystem namespace.

`chroot3` and `chroot4` have quite a few dependencies. They look for the system's canonical hostname using the Name Service Switch, so you'll need at least `/lib/libnss_files.so.2` and `/etc/hosts`, and possibly other files as well if you need other name services than flat files to get your system's canonical hostname (this shouldn't have been the case on our vservers). They also invoke `/bin/date`, which has some interesting dependencies – it links `/lib/libpthread.so.0` and `/lib/librt.so.1` in addition to the usual dynamic linker and C library, and needs `/etc/localtime` to get the timezone right.

This is a good example of how a `chroot()` jail quickly gets more cluttered as the program being jailed gets more complicated. As the security benefit of the jail is reduced as it gets more populated, whether a program is worth `chroot()` jailing needs some consideration.

7. *Packet sniffing. Optional.* Go capture some packets on your favorite network. Analyze the traffic streams, and point out security weaknesses. Suggest ways to improve the security of the traffic going over the network.

8. *Forensic analysis. Optional.* Try problems 1-7 of the Honeynet Project's Forensic Challenge `<http://www.honeynet.org/challenge/>`, except that I don't expect you to disassemble the malware to figure out what it does. This is an old challenge, and the image represents a fairly out-of-date system, but the techniques still apply in investigating break-ins today.

See the Honeynet Project's analysis:

<http://www.honeynet.org/challenge/results/dittrich/>