Advanced Unix System Administration
Spring 2008
Homework 2 Solutions

1. *POSIX ACLs, and portability considerations.* I noted in class that POSIX draft ACLs don't necessarily behave the same way on all the systems where they're implemented. While the behavior of the ACL access checking doesn't differ the way I implied it did in class, there are plenty of user-visible differences in the utilities. Here's a chance to work with ACLs a bit, and look at portability differences between Linux and Solaris (specifically, Solaris Nevada build 78, a pre-release version of Solaris 11).

   a. On a Linux box (like `tempest`), create a new file with permissions `0600`, then set ACLs on it allowing user `nobody` to read it (don't forget the mask). Look at the output of `ls -l` on the file. Repeat on Solaris. What differences, if any, do you notice?

   On Linux:

```
sluo@tempest:~$ touch file
sluo@tempest:~$ chmod 600 file
sluo@tempest:~$ setfacl -m u:nobody:4,m:4 file
sluo@tempest:~$ ls -l file
-rw-r-----+ 1 sluo sluo 0 2008-04-21 19:21 file
```

   On Solaris (I'm using a Solaris 10 box, but you should have gotten the same results):

```
[sluo@asteroid ~]$ touch file
[sluo@asteroid ~]$ chmod 600 file
[sluo@asteroid ~]$ setfacl -m u:nobody:4,m:4 file
[sluo@asteroid ~]$ ls -l file
-rw-------+  1 sluo     ocf            0 Apr 21 19:22 file
```

   The commands work the same way, but `ls` shows different permissions for the file. Recall that the mask attribute for a POSIX ACL is actually stored as the regular group permissions on the file, and not as part of the ACL list; the Linux `ls` is simply displaying the regular permission bits, whereas the Solaris `ls` is smart enough to look at the ACLs on the file:

```
[sluo@asteroid ~]$ setfacl -m g::4 file
[sluo@asteroid ~]$ ls -l file
-rw-r-----+  1 sluo     ocf            0 Apr 21 19:22 file
```

   In both cases, a `+` is displayed at the end of the permissions string, to indicate that an ACL is set on the file.

(In either case, you could have used symbolic modes:

```
setfacl -m u:nobody:r--,m:r-- file
```

would also have worked. I'm just used to numeric modes.)

b. Remove the ACLs on the file, on both Linux and Solaris. What did you do differently on each system?

   Linux:

```
sluo@tempest:~$ setfacl -x u:nobody,m file
sluo@tempest:~$ ls -l file
-rw------- 1 sluo sluo 0 2008-04-21 19:21 file
```

   Solaris:

```
[sluo@asteroid ~]$ setfacl -d u:nobody,m: file
[sluo@asteroid ~]$ ls -l file
-rw-------   1 sluo     ocf            0 Apr 21 19:22 file
```

   Not only do they need different arguments to `setfacl`, they parse the ACL specification list differently! Annoying.

c. Now create a directory with permissions `0700`. Set a default ACL on it allowing user `nobody` to read files created in this directory, and `cd` into and read subdirectories of this directory. How does the procedure for doing this differ?

   Linux:

```
sluo@tempest:~$ mkdir dir
sluo@tempest:~$ chmod 700 dir
sluo@tempest:~$ setfacl -m d:u:nobody:5,d:m:5 dir
```

   Solaris:

```
[sluo@asteroid ~]$ mkdir dir
[sluo@asteroid ~]$ chmod 700 dir
[sluo@asteroid ~]$ setfacl -m \
    d:u::7,d:g::0,d:o:0,d:u:nobody:5,d:m:5 dir
```

   Notice you have to specify all the default ACL entries – if you'd tried what you did on Linux, you'd see:

```
[sluo@asteroid ~]$ setfacl -m d:u:nobody:5,d:m:5 dir
Missing user/group owner, other, mask entry
aclcnt 6, file dir
```

   When default ACL entries are present, the user and group owner and other entries must always be present; unlike the Linux `setfacl`, the Solaris `setfacl` does not fill these values in for you with sane defaults if you omit them.

d. Create a new file and a new subdirectory in this directory, and look at the ACLs that they inherit. Are they the same on both Linux and Solaris? How does the mask interact with the ACL entry for user `nobody`?

Linux:

```
sluo@tempest:~$ cd dir
sluo@tempest:~/dir$ touch file
sluo@tempest:~/dir$ mkdir dir
sluo@tempest:~/dir$ getfacl file
# file: file
# owner: sluo
# group: sluo
user::rw-
user:nobody:r-x                 #effective:r--
group::---
mask::r--
other::---

sluo@tempest:~/dir$ getfacl dir
# file: dir
# owner: sluo
# group: sluo
user::rwx
user:nobody:r-x
group::---
mask::r-x
other::---
default:user::rwx
default:user:nobody:r-x
default:group::---
default:mask::r-x
default:other::---
```

Solaris:

```
[sluo@asteroid ~]$ cd dir
[sluo@asteroid ~/dir]$ touch file
[sluo@asteroid ~/dir]$ mkdir dir
[sluo@asteroid ~/dir]$ getfacl file

# file: file
# owner: sluo
# group: ocf
```

```
user::rw-
user:nobody:r-x         #effective:r--
group::---              #effective:---
mask:r--
other:---
[sluo@asteroid ~/dir]$ getfacl dir

# file: dir
# owner: sluo
# group: ocf
user::rwx
user:nobody:r-x         #effective:r-x
group::---              #effective:---
mask:r-x
other:---
default:user::rwx
default:user:nobody:r-x
default:group::---
default:mask:r-x
default:other:---
```

The inherited ACLs are (thankfully) the same on Linux and Solaris. Notice the ACL on the file you created; the ACL for `nobody` says that read and execute permissions should be available, but the mask no longer has the execute bit set (presumably because `touch` created the file with requested permissions 0666, which prevents the file from being executed), so execute is not permitted.

e. Use `chmod` to add group execute permissions to the file you created in part (d). Look at the ACLs now; why did they change in this way? Compare the ACLs on Linux and Solaris. Are they the same?

Linux:

```
sluo@tempest:~/dir$ chmod g+x file
sluo@tempest:~/dir$ getfacl file
# file: file
# owner: sluo
# group: sluo
user::rw-
user:nobody:r-x
group::---
mask::r-x
other::---
```

Solaris:

```
[sluo@asteroid ~/dir]$ chmod g+x file
[sluo@asteroid ~/dir]$ getfacl file

# file: file
# owner: sluo
# group: ocf
user::rw-
user:nobody:r-x         #effective:r-x
group::--x              #effective:--x
mask:r-x
other:---
```

In both cases, execute permission is added to the mask (remember that the mask is stored as the regular group permission). Solaris `chmod` also creates a group owner ACL for you with the execute bit set, which results in more intuitive behavior than on Linux (where `chmod g+x` doesn't give the group owner execute permission, like it normally would). In both cases, though, the addition of execute permission to the mask also allows user `nobody` to execute the file – perhaps not the result you were expecting!

f. Set `umask 777`, then try creating a new file. Is the `umask` honored?

Linux:

```
sluo@tempest:~/dir$ umask 777
sluo@tempest:~/dir$ touch file2
sluo@tempest:~/dir$ getfacl file2
# file: file2
# owner: sluo
# group: sluo
user::rw-
user:nobody:r-x                 #effective:r--
group::---
mask::r--
other::---
```

Solaris:

```
[sluo@asteroid ~/dir]$ umask 777
[sluo@asteroid ~/dir]$ touch file2
[sluo@asteroid ~/dir]$ getfacl file2

# file: file2
# owner: sluo
# group: ocf
user::rw-
```

```
user:nobody:r-x          #effective:r--
group::---               #effective:---
mask:r--
other:---
```

In a word, no. This is rather unfortunate, as it means, for example, that setting a default ACL on your home directory is not such a good idea (because programs which change the umask and expect the files they create to be private won't have such behavior).

g. Trace the `getfacl` command on both systems (on Solaris, you want `truss(1)`). Identify the system call used to access the ACL list. Is it the same on both systems? *Optional:* What does this reveal about the way POSIX draft ACLs are implemented on the two systems?

   On Linux, `strace getfacl file` contains the following:

```
getxattr("file", "system.posix_acl_access", "\x02\x00"..., 132)
    = 44
```

   Linux stores POSIX ACLs as a form of extended attribute in the underlying filesystem, so the system call used to get the ACLs is the generic one for accessing extended attributes. A `libacl` userspace library provides functions so that programs don't have to know this.

   On Solaris, `truss getfacl file` gives us:

```
acl("file", GETACLCNT, 0, 0x00000000)          = 5
acl("file", GETACL, 5, 0x08063778)             = 5
```

   Solaris has a dedicated `acl()` system call for accessing ACLs. (This doesn't tell us anything about how ACLs are implemented on Solaris, by the way.)

These sorts of portability differences drive everyone who has to work on multiple systems crazy; this is why, where possible, you want to demand that your software vendors work towards creating cross-platform standards, and adhere to existing standards to the greatest degree possible.

2. *A login session.*

   a. Trace an SSH login. What UID does the login process initially run as? Must this be the case, and why? Identify the parts of the output that show when the process changes user and group IDs, and when your login shell is invoked.

      `sshd` initially runs as `root`. On a standard Unix system, this must be the case, since `sshd` is going to have to eventually change users and become you for your login session.

      There are two places where `sshd` changes UID or GID for a password-based login; only one of them is absolutely necessary, the rest being done for security reasons. The first time is shortly after `sshd` forks a child process to handle the incoming network connection from an unauthenticated user:

```
12505 setregid(65534, 65534)            = 0
12505 setreuid(103, 103)                = 0
```

On Linux, at least, this is sufficient to change real, effective, and saved UIDs and GIDs (see the man page) to the new values (GID 65534 corresponds to `nogroup`, and UID 103 is `sshd` on this system). Hence the process directly handling the network connection has dropped all its privileges, which reduces the impact an attacker can have if he compromises this daemon.

The other time is when actually performing the login:

```
12508 setregid(1000, 1000)              = 0
12508 setreuid(1000, 1000)              = 0
```

This changes UID and GID to my UID and GID, in preparation for logging me in.

The actual running of my login shell comes a bit later:

```
12509 execve("/bin/bash", ["-bash"], [/* 11 vars */]) = 0
```

Note the "-" at the beginning of the first element of the arguments list (remember, by convention, this is the name of the process); this tells the shell that it is to run as a login shell.

b. Look up what a POSIX "session" is, and what a "session leader" is. Give two reasons why it's important that a new session be created for a user login. (Hint: one has to do with an important feature people expect from their shells; another has to do with what happens if, say, the SSH daemon dies or is killed.) Identify in the trace output where the session corresponding to your login shell is created. Identify which process ends up being the session leader for the login session once the login shell is invoked. (If you're having trouble finding these definitions, try looking in the Single Unix Specification, available from the Open Group. I encourage you to look for other resources first, though, since, like all standards documents, the writing is extremely dense.)

The Single Unix Specification version 3 (SUSv3) defines a "session" as (Base Definitions volume, §3.337)

> A collection of process groups established for job control purposes. Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership; see `setsid()`. There can be multiple process groups in the same session.

§3.338 defines a session leader as "A process that has created a session".

As is unfortunately not too uncommon for standards documents, this is a definition that only makes sense if you already know what a session is. The rationale section for the documentation on `setsid()` (System Interfaces volume) has more useful information:

> The `setsid()` function is similar to the `setpgrp()` function of System V. System V, without job control, groups processes into process groups and creates new process groups via `setpgrp()`; only one process group may be part of a login session.
>
> Job control allows multiple process groups within a login session. In order to limit job control actions so that they can only affect processes in the same login session, this volume of IEEE Std 1003.1-2001 adds the concept of a session that is created via `setsid()`. The `setsid()` function also creates the initial process group contained in the session. Additional process groups can be created via the `setpgid()` function. A System V process group would correspond to a POSIX System Interfaces session containing a single POSIX process group. [...]

In other words, each "job", for purposes of process control, is a process group, and these are all grouped into your shell's session. When your shell wants to suspend or resume a job, it sends the appropriate signal (usually `SIGTSTP` for a suspension, and `SIGCONT` for a continuation) to the corresponding process group (this just sends that signal to all the processes in a process group). Hence one of the important reasons to create a new session for your user login is to ensure that job control works correctly.

The other reason to create a new session is that this ensures that your login process is in a different session and process group than the main SSH daemon. This way, if the SSH daemon is killed, your login isn't killed off as well; this allows an administrator to restart SSH without forcibly logging users off.

With all of this, it should be clear what to look for in the trace output:

```
12509 setsid()                              = 12509
```

This is called before my login shell is invoked, which means, that, once the login shell is `exec()`ed, it is the session leader. Given the above discussion, none of this should come as a surprise.

c. Look up what a "controlling terminal" is. Why is it important that a user login has a controlling terminal? Identify in the trace output when the login process acquires your controlling terminal, and give the name of the device file which corresponds to it.

The Single Unix Specification version 3 (SUSv3) defines a "controlling terminal" as (Base Definitions volume, §3.114)

> A terminal that is associated with a session. Each session may have at most one controlling terminal associated with it, and a controlling terminal is associated with exactly one session. Certain input sequences

from the controlling terminal cause signals to be sent to all processes in the process group associated with the controlling terminal.

In other words, the controlling terminal of a session is the terminal that that login session is attached to. A full description of the terminal interface is given in Base Definitions volume §11, "General Terminal Interface". Roughly, a terminal provides interactive input and output, and provides some features such as interpreting Ctrl-Z as "suspend foreground process" and killing off processes in its login session if it's disconnected (user logs out, or SSH connection or serial line drops, for example). It's important for a login to have a controlling terminal because many programs, particularly interactive command-line programs, expect to have a terminal available.

SUS does not specify how one goes about acquiring a controlling terminal after having lost it with `setsid()`, but at least on Linux and several other systems, this is done by opening the device file for the terminal:

```
12509 open("/dev/pts/2", O_RDWR)        = 8
```

This happens after the new session is created, but before the login shell is invoked – again, not too surprising, given what we know.

d. Outline the steps needed for a running process (such as `sshd` or `login`) to create a normal login session for a user. What is the latest point at which you could effectively set up resource limits for a login session, and why?

   The process roughly goes as follows (assuming the user has been authenticated already):

   - Change user and group IDs to those of the user being logged in.
   - Fork, to ensure that the following call to `setsid()` succeeds (`setsid()` requires that the the calling process is not process group leader, and doing a `fork()` ensures that it isn't).
   - Call `setsid()` to become session leader of a new session.
   - Open a controlling terminal.
   - Change the current working directory to the user's home directory.
   - Run the user's login shell.

   It's possible to set up resource limits at any point before the user's login shell is run. Resource limits are inherited by children of a process, so it's safe to set them even before forking, and any limits set before the login shell is run will be inherited by any process the user runs from that shell. On the other hand, if the limits are set after the user's shell is run, he/she has the opportunity to launch processes which are not subject to the limits.

3. *User names and UIDs.* As I mentioned in class, the relationship between user names and UIDs is not as absolute as you might believe.

a. Create a new user, and observe what `ls` shows the owner of that user's home directory to be.

```
tempest:~# adduser foobar
[...]
tempest:~# ls -ld /home/foobar
drwxr-xr-x 2 foobar foobar 4096 2008-04-22 02:51 /home/foobar
```

No surprises so far.

b. Change the username (edit `/etc/passwd` and `/etc/shadow`). Does the owner displayed by `ls` for the user's home directory change as well?

After changing the username to `baz` by editing the appropriate files:

```
tempest:~# ls -ld /home/foobar
drwxr-xr-x 2 baz foobar 4096 2008-04-22 02:51 /home/foobar
```

c. Remove the `/etc/passwd` entry for your new user (make a backup first, you'll need it for the rest of the problem). Now what does `ls` show for the ownership of the user's home directory?

Now we see:

```
tempest:~# ls -ld /home/foobar
drwxr-xr-x 2 1007 foobar 4096 2008-04-22 02:51 /home/foobar
```

d. Based on the above, how do you think the ownership information is stored on disk? How do you think `ls` decides what username to display?

Clearly, ownership information is stored on disk with the numeric user ID – which is in keeping with the general principle that access checks on users are done on the user ID, not the username. On most systems, `ls` decides what username to display for a particular UID based on the mapping defined in `/etc/passwd`; more generally, the Name Service Switch is used to determine what data source to use for username-UID mappings.

If, for some reason, you managed to do these exercises on a machine where the name service cache daemon (`nscd`) was running, it might have taken several minutes for the cached entries to expire; hence the changes to `/etc/passwd` may not have been reflected immediately in the `ls` output. `nscd` is rare on systems that don't use network NSS sources (LDAP, NIS, etc.), though, so this wasn't likely a problem for you.

e. Restore the entry you removed, and create a second entry in `/etc/passwd` with the same UID, but a different username. Copy the `/etc/shadow` entry for the first user and change the username to match your duplicate user. Try logging in as both. Do things look and behave the same for both? Is there a difference in what files they can read? Can they kill each other's processes?

You should notice that everything looks and behaves the same way for both users, that they can read the same files, and kill each other's processes. Again, remember that in general, access checks in the kernel are done by UID and not by username, so this makes sense.

f. Add one of the users to a group to which it doesn't already belong, and try logging in as both users again. Do they both belong to this group? Change one of the users' passwords. Do they both have the same password now?

   Having added `baz`, but not `quux`, to group `cdrom` by editing `/etc/group`, we see this for `baz`:

   ```
   baz@tempest:~$ echo $USER
   baz
   baz@tempest:~$ groups
   foobar cdrom
   ```

   For `quux`, on the other hand:

   ```
   baz@tempest:~$ echo $USER
   quux
   baz@tempest:~$ groups
   foobar
   ```

   In other words, it is possible to add one user to groups without adding the other.

   You should also have noticed that changing the password for one user does not affect the password for the other user.

   These are two of the places where the information is indexed by username and not UID, so this isn't too surprising; the inconsistency is perhaps a bit troubling, but there's really nothing to be done about it at this point . . .

g. Based on the above, why might you want to have two users with the same UID? How might this be abused?

   Having two users with the same UID might be useful in a situation where you want the same "user" to be able to have two different group lists; for example, where you normally want to log in without certain extra groups, but want to be able to get those groups' privileges without losing access to files owned by that user. On the other hand, a second account with UID 0 might go completely unnoticed, but still have full root privileges (attackers have been known to create extra accounts with UID 0 as a form of crude back door, in fact).

4. *A simplified set of init scripts.* I mentioned in class that reading init scripts is one of the best ways to learn how an unfamiliar, inadequately documented system is put together and how to configure it. Here's a simplified set of init scripts that does what it takes to configure and boot the system, while being easier to read through than real scripts (which have to deal with many varied configurations, errors, etc.). Note

that, to avoid complexity, the configuration looks different than most traditional Unix systems.

a. Examine `/etc/inittab`. Do these init scripts implement a SysV- or BSD-style init? How do you know?

   They implement SysV-style init – they deal with runlevels, which is a concept traditional BSD-style init doesn't have.

b. Describe in general terms the tasks that will be performed on every system boot, no matter which runlevel is selected.

   `/etc/inittab` has this to say about what happens when the system boots:

   ```
   si::sysinit:/etc/init.d/rc S
   ```

   On boot, then, `/etc/init.d/rc S` is run. Reading this script, we find that this just runs all the scripts linked to in `/etc/rcS.d`, with links starting with S called with the `start` argument and links starting with K called with `stop`:

   - `S01hostname`: sets the system hostname to the contents of `/etc/hostname`
   - `S02mountkernfs`: mounts `/proc` and `/sys`, pseudo-filesystems used by various programs on Linux
   - `S10mountroot`: checks the root filesystem for errors, then mounts it
   - `S15modules`: loads kernel modules listed in /etc/modules
   - `S20mountfs`: check and mount the rest of the filesystems
   - `S30networking`: configure network interfaces
   - `S50wtmp`: create `/var/run/utmp`. `utmp` is used by the system to record who's currently logged in.
   - `S70urandom`: load some stored entropy (randomness) from the last boot into the kernel random number generator, to ensure that the generated random numbers won't be predictable

c. How would you configure a new filesystem (`/dev/hdc1`, type `ext3`, mounted on `/export` with options `nodev` and `nosuid`) to be mounted on boot? What if you didn't want it to be mounted on boot? How would you change the mount options for the root filesystem? How would you force the system to check all filesystems on the next boot?

   To add the new filesystem, add a file in `/etc/filesystems` (for example, `/etc/filesystems/3`, though any other name would do; note that filesystems are mounted in the order which these filenames sort), with the following contents:

   ```
   DEVICE=/dev/hdc1
   FSTYPE=ext3
   MOUNTPOINT=/export
   FSOPTS=nodev,nosuid
   MOUNT_AT_BOOT=yes
   ```

To prevent this from being mounted on boot, simply change the value of MOUNT_AT_BOOT to any value other than "yes" (the logical choice being "no", but anything else would do too). (Note that it won't do to simply remove MOUNT_AT_BOOT from the file, since this results in behavior that depends on what the previously considered filesystem was configured for! This is a bug in the script, which I didn't think was worth fixing.)

To change the mount options for the root filesystem (or any other filesystem, for that matter), edit the appropriate config file file (for the root filesystem, /etc/filesystems/0) and change or add the FSOPTS line as necessary.

To force a filesystem check on the next boot, touch /forcefsck.

Notice that this is nothing like the usual filesystem configuration, which involves editing /etc/fstab (or a similar file). The init scripts can implement any configuration interface they like; it's just that, in most cases, distributors implement standard interfaces like fstab. I chose not to do this here because the code needed to parse the file would have been more complex and difficult to read.

d. How would you add a new network interface (device name eth1, with IP address 10.20.42.42, netmask 255.255.255.0)? How would you change eth0's IP address? How would you change the default route?

   To add a new network interface, add a new file /etc/netif/eth1 with the following contents:

```
IPADDR=10.20.42.42
NETMASK=255.255.255.0
```

   To add another interface, just create a file in /etc/netif with the name of the interface, containing the appropriate settings.

   To change eth0's IP address, edit the IPADDR variable in /etc/netif/eth0.

   To change the default route, change the contents of /etc/defaultrouter.

   As far as I know, this network configuration style isn't shared by any serious distribution, though elements of it are inspired by network configuration on the BSDs and Solaris.

e. What is the default runlevel? Describe in general terms the tasks that are performed when booting into this runlevel.

   /etc/inittab contains the following:

```
id:3:initdefault:
```

   which tells us that runlevel 3 is the default.

   To determine what happens when we boot into this runlevel, we look at the lines in /etc/inittab with a "3" in the runlevel field:

```
r2:23:wait:/etc/init.d/rc 2
```

```
r3:3:wait:/etc/init.d/rc 3
1:23:respawn:/sbin/getty 38400 tty1
```

In other words, when booting into runlevel 3, first `/etc/init.d/rc 2` is run, then `/etc/init.d/rc 3`, then `/sbin/getty 38400 tty1` is run and respawned whenever it dies. This is a SysV-style init in the way Solaris implements it; when booting into runlevel N, tasks for runlevels 2 through $(N-1)$ are run as well.

The startup scripts in runlevel 2, which are run first, are:

- `S10sysklogd`: starts the system log daemon
- `S11klogd`: starts the kernel log daemon (reads from the kernel message log and writes the contents to syslog)
- `S89cron`: starts the cron daemon

Next, the script in runlevel 3 is run:

- `S20ssh`: starts the SSH daemon

The `getty` then provides terminal login services on `tty1`, the first virtual console. Each `getty` only handles one login session; once a user logs out, the `getty` dies and is respawned by `init`.

f. Describe in general terms what tasks are performed when the system is shut down. (Hint: which runlevel corresponds to shutdown?)

Remembering that the system goes into runlevel 0 when it's shut down, we look in `/etc/inittab` and find that the scripts in `/etc/rc.0` are executed on shutdown:

- `K10sshd`: stop the SSH daemon
- `K39cron`: stop the cron daemon
- `K48klogd`: stop the kernel log daemon
- `K49sysklogd`: stop the system log daemon
- `K50killall`: kill all processes still running on the system
- `K55wtmp`: write a reboot record into `/var/log/wtmp` (which records everyone who's logged in, and each system start and stop)
- `K55urandom`: save some entropy to disk to seed the kernel random number generator with at the next boot
- `K60networking`: bring down network interfaces
- `K80mountfs`: unmount filesystems (note that, for simplicity, this doesn't deal with mounted filesystems which aren't in `/etc/filesystems`! this is probably a bug)
- `K90mountroot`: remount the root filesystem read-only
- `K99poweroff`: turn off the computer

g. Suppose you wanted to have a webserver (init script `/etc/init.d/httpd`) be started on system startup and stopped gracefully on system shutdown. How would you set this up?

   Add some links pointing to `/etc/init.d/httpd` in the appropriate runlevels:

   - `/etc/rc3.d/S20httpd`
   - `/etc/rc0.d/K10httpd`
   - `/etc/rc6.d/K10httpd`

   Where the start script is placed in the boot sequence doesn't really matter, as long as it's after filesystems have been mounted and the network has been brought up. The placement of the shutdown scripts doesn't really matter either, as long as the server's stopped before it's killed off via brute force by the `killall` script.

   To be complete, we should also add a `/etc/rc1.d/K10httpd` killing off the server when we go into single user mode, but I didn't ask for that in the problem …