Advanced Unix System Administration
Spring 2008
Homework 4

This assignment is due via email to <sluo+decal@ocf.berkeley.edu> by 11:59 PM on **Wednesday, May 14**. (More realistically, any time between now and when I have to submit grades will do.) All the files mentioned are in `~sluo/hw4-files` on the login server (`tempest.ocf.berkeley.edu`), and in a tarball `hw4-files.tar.gz` available from the website.

1. *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 executable* 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 and/or groups, if appropriate.

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

   c. Which design do you believe to be more secure? Why?

2. *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.

   For this exercise, you have the use of a container, located at the IP `10.20.4.x1`, where `x` is the last digit of your UID on the login server; as usual, log in with your login server username and password.

   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?

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.

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.

You've no doubt observed that, as a program gets more complex, it becomes more difficult to `chroot()` jail, and you end up copying more of the host filesystem into the chroot to make it work. As having more in the chroot reduces the security benefit, there is a definite cost-benefit tradeoff to building a `chroot()` jail – simple apps can usually benefit, whereas complex ones with lots of filesystem dependencies (especially if those dependencies include things like shells and/or language interpreters) may not be worth chrooting. Daemons that know how to `chroot()` themselves (or parts of themselves) can be better in this regard, as the authors can design the chroot routine to work around most of these difficulties.

A `chroot()` jail only provides filesystem isolation. For stronger isolation of your daemons, you'll need to look into system-specific features. FreeBSD has the `jail()` system call, providing process, network, and user isolation; see the `jail(8)` and `jail(2)` man pages for details. Linux kernels with the Linux-VServer patch provide security contexts, which in conjunction with `chroot()` and judicious use of the capabilities system, provide even stronger guarantees; see the `chcontext(8)` man page from the util-vserver distribution and other Linux-VServer documentation for details. These technologies, as well as some others (OpenVZ for Linux, Solaris zones) which can't be set up to confine just one process (as far as I know), can be used to create complete systems as well. This is in fact probably their most common use – with the additional isolation, breaking out of the container becomes much harder (should be impossible, really, without exploiting a kernel bug of some sort), so there's less benefit to making life more painful by building a extreme minimalist environment.