Advanced Unix System Administration
Spring 2007
Homework 3-4

This assignment is due via email to <sluo+decal@ocf.berkeley.edu> by 11:59 PM on **Friday, April 27** (note that you have two weeks to do this assignment). All the files mentioned are in `~sluo/hw3-files` on the login server (`plague.ocf.berkeley.edu`), and in a tarball `hw3-files.tar.gz` available from the website. If you do not already have access to the login server, you need to email me about setting up an account. This is a long assignment; you don't want to wait until the last minute to get started!

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.

   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. *Note:* you do not need to write out each packet byte-by-byte – a thorough description of the header and contents of each frame will do.

   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. *Note:* you do not need to write out each packet byte-by-byte – a thorough description of the header and contents of each frame will do.

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.

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?

   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? *Optional:* If you have access to a suitable zombie host and a machine which you can do network configuration on (not your scanning host!), verify this.

   c. What can you do to prevent idle scans from being launched from inside 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. (Driving up the system load might help make your exploit work more reliably. If you do this, please clean up when you're done!)

   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?

   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`?

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?

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.

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.

   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.

   c. Which design is more secure? Why?

6. *Setting up a `chroot()` jail. This exercise must be done from the login server.* 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 `10.20.2.xx`, where `xx` is the last two digits of your UID on the login server plus 10; for example, I have UID 1000, so my container would be `10.20.2.10`. An account with your username on the login server and the files you need has been created; both it and the root account for the container have their passwords set to your password on the login server. Logins are via SSH, as usual, with direct root logins disabled. The containers don't have very much at all installed (Debian packages of priority important and higher, GCC, OpenSSH, `strace`, and `ltrace`), and have strict resource limits (32 MB maximum memory usage, 128 processes running); they won't stay around for long past this assignment's due date.

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

containers) 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.

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.