CS155: Computer Security **Spring 2002**

Project #2

Due: Monday, May 27th, 2002.

Summary In this project, you will write Josh, the journaling operator shell. Josh is a setuid-root shell that allows users to undertake a subset of root's capabilities.

Your starting point is OSH, a minimal shell developed by Gunnar Ritter.

You may work alone, or in pairs. You should not collaborate with others outside your group.

You will use the Boxes system again. Remember to test your Josh in a closedbox.

The Problem As discussed in class, the classical Unix permissions model does not manage privilege delegation effectively. To perform routine administrative tasks (such as cleaning out the /tmp directory), one must possess the root password, and can therefore perform other, undesirable actions (such as reading users' mail).

This limitation manifests itself most acutely in large Unix installations, where there are typically many administrators tasked with various duties.

Various systems have been devised to overcome this limitation, and allow unprivileged users to undertake some subset of root's capabilities. A commonly-used example is sudo (http://www.sudo.ws/).

We will take an approach more like that taken by OSH, the Operator Shell (`http://www.engarde.com/~mcn/osh.html`). You might want to read the SANS III conference paper linked from the OSH home page.

Our Solution We will develop a new shell, called Josh — the Journaling operator shell — which will enable privilege delegation in the Unix environment.

In Unix, the command interpreter is not part of the kernel, and is in fact modular and interchangeable. The command interpreter program is called a shell; it reads user requests, parses them, and makes the system calls required to fulfill the requests on the user's behalf. Seventh Edition Unix (Jan 1979) shipped with the Bourne Shell, /bin/sh; 2BSD (May 1979) shipped with Bill Joy's C Shell (/bin/csh). The KornShell (/bin/ksh) was first released in 1983 and was for some time an at-cost addon for Unix System V.

Today, people typically use either bash, a Bourne-shell reimplementation, or tcsh, a C Shell enhancement.

None of these shells is designed to be run setuid-root.

Starter Code Writing a shell is an excellent example of a Simple Matter of Programming (`http://tuxedo.org/jargon/html/entry/SMOP.html`). Besides the fork-exec-open-pipe infrastructure required for implementing shell pipelines, there's a considerable amount of parsing and bookkeeping that must be taken care of.

To keep you from having to start your project by implementing a shell — an excellent CS193u project, by the way — we provide you with OSH, the Old Shell (no relation to the Operator Shell). OSH is a feature-for-feature-compatible reimplementation of the Sixth Edition shell (May 1976) by Gunnar Ritter (`http://omnibus.ruf.uni-freiburg.de/~gritter`).

While OSH lacks some features we expect in modern shells (notably backticks and control structures) it packs quite a few features into 837 lines of C. (Bash is about 100,000 lines.)

Recommended Reading For some practical ideas about security dos and don'ts, read David Wheeler's "Secure Programming for Unix and Linux HOWTO" (`http://www.dwheeler.com/secure-programs/`), which is linked from the course website.

For Unix programming in general, there is no better source than W. Richard Stevens' "Advanced Programming in the UNIX Environment" (Addison-Wesley, 1992, ISBN 0-201-56317-7). Go buy it now.

You might also want to refer to the source of various software components with which Josh will interact, such as the Linux kernel (`http://www.kernel.org/`) or the GNU C Library (`http://www.gnu.org/software/libc/`).

Using Boxes The Boxes distribution has been upgraded to the latest User-Mode Linux patch. This patch should hopefully address some of the instability that was encountered during the first programming project.

The filesystem image has been upgraded to the latest from Debian. We have added the Expect utility (`http://expect.nist.gov/`), to allow you to test your shell out non-interactively, if you wish.

The wrapper interface to Boxes remains the same. You should check out the FAQ posted to the newsgroup for basic information on getting Boxes up and running.

Step One: Secure the Perimiter Now we describe the tasks you must undertake in creating Josh. Of course, you need not stick to the order in which we present them.

Though OSH, your starter shell, is good code, it was not written to be run setuid root.

You should start by familiarizing yourself with the structure of the shell, auditing it with security in mind, and thinking about how best to extend it.

Step Two: Executables Josh allows users to run some programs that otherwise they could not. The file that controls this behavior is `/etc/josh_exec`. This file (which should be installed `root:root`, mode 600) has entries, one per line, in the following format:

```
userid:progpath
```

(Without the initial indentation.) Here, `userid` is a user's login name; progpath is some absolute path to a program. Any program listed in `josh_exec` for a particular user should be executed as root, not as the user, whenever it is invoked. For example, if `/etc/josh_exec` includes the line

```
alice:/bin/kill
```

Then any kill invoked by alice should run as root.

Note that your Josh will have to figure out which program is actually invoked when a user types a non-absolute file name, or relies on the value of `$PATH` to find the program.

Your Josh should ignore any entries in `josh_exec` that do not name executable programs.

Step Three: File Access An administrator will need to read some files that are not readable by all users, and possibly to write to other files that are not writable by all users.

Josh will open these files on behalf of the administrator.

Josh decides whether to open these files based on a configuration file, `/etc/josh_access`. This file (which should be installed `root:root`, mode 600) has entries, one per line, in following format:

```
userid:filepath:perms
```

Here, `userid` is a user's login name; `filepath` is some absolute file name; and `perms` is of the form `[+-](r|w|rw)`. A + grants a positive right; a - takes away a previously granted right. For example, the entry

```
alice:/etc/motd:+w
```

Means that Alice is allowed to write to the file `/etc/motd`. (Read access need not be granted, because everyone can read `/etc/motd` by default.)

Entries are cumulative. Thus the two lines

```
bob:/etc/ssh/ssh_host_key:+r
bob:/etc/ssh/ssh_host_key:+w
```

Together mean that Bob can both read and write to the `ssh_host_key` file.

Negative rights are only meaningful in canceling a previous access grant: If a user has read or write permissions to some file without Josh, then a negative `josh_access` entry should not prevent her from accessing the file.

If the filepath specifies a directory, then the read/write permission is to apply recursively to all files under that directory. For example, the lines

```
alice:/etc:+r
alice:/etc/shadow:-r
```

Mean that Alice is allowed to read all files in `/etc` (and subdirectories thereof), except `shadow`. Note that you should not allow write access to the directories themselves; that usually is too much power to delegate.

Your Josh should deal with these access permissions correctly for files named by shell redirections. For example, files for which read access has been granted can be redirected from (using `<`); files for which write access has been granted can be redirected to (using `>`).

Step Four: Editor To allow Alice to edit some system files, it would be nice simply to add a line enabling her to run, say, `vi` in `josh_exec`. But then she could modify any file on the system, leading to a root compromise.

Instead, we could write a shell script that invokes `vi` on a particular file (compare BSD's `/usr/sbin/vipw`), and give Alice execute permission to that.

Unfortunately, all editors commonly available for Unix allow one to choose to write to files other than the one originally opened; running an editor as root – even with a specially-chosen command line – would precipitate an immediate root compromise.

So, we must go about editing files in a roundabout way. We describe one approach. First, one copies the file to be edited to some temporary location; then, one allows the user to edit the temporary copy of that file, without special privileges; finally, one copies the edited file back into place.

Implement a new shell builtin, "`edit`", that allows secure file editing. Your edit command should launch the editor named in the `$EDITOR` environment variable, or, failing that, `/usr/bin/vi`.

Step Five: Journaling Josh is the Journaling Operator Shell because it keeps journals of users' activities for auditing purposes.

Your Josh should use `syslog` (facility `authpriv`, level `info`) to record each command run by a Josh user. Your log line should contain the user's login name, the full command line, and either "`OK`" or "`DENIED`", separated by spaces.

Step Six: Extra Credit Josh is not only an excellent system-administration tool, but also a collaboration utility.

For extra credit, allow users to create `~/.josh\_exec` and `~/.josh\_access`, in the same format as root's Josh files in `/etc`. Programs listed in a user's `.josh_exec` file should run as that user; files lister in a user's `.josh_access` file should be accessed as that user.

Josh should look for user-granted privileges only when its working directory is the affected user's home directory. In other words, if Alice is running Josh, and wishes to access one of Bob's files, Josh should examine `~bob/.josh\_access` only when Alice's working directory is `~bob`.

Also, root-granted privileges in `/etc/josh*` should always override user-granted ones.

Deliverables As in the first programming assignment, you will use the online Leland submit script, `/usr/class/cs155/bin/submit`. This is `pp2`.

The directory which you submit must contain the `Makefile` and all source files necessary for building your Josh when the `make` command is issued.

Along with your shell, you must include file called `ID` which contains, on a single line for each person in your team, the following: your SUID number; your Leland username; and your name, in the format last name, comma, first name. An example:

```
$ cat ./ID
3133757 binky Clown, Binky The
$
```

If you work alone, `ID` should have exactly one line. If you work with a partner, `ID` should have exactly two lines.

Do not include any identifying information in any file except `ID`.

You will also want to include a `README` file with details of your design, comments about your experiences, or suggestions for improving the assignment.

How You Will Be Graded All grading will take place in a closedbox environment, with Josh installed setuid root as `/usr/local/sbin/josh`.

We will run Josh interactively, using expect. We will grade your Josh on functionality. Why? A program that does nothing is easy to secure; the difficult task is to achieve both functionality and security.

In the next phase, we will randomly assign to each group a Josh written by another. (This is why it is important that you not identify yourselves in any file other than `ID`.) Then, you will audit the Josh you are assigned, looking for security vulnerabilities.

What Constitutes a Security Vulnerability In an application like Josh, a security vulnerability is anything that allows a player in the system to obtain additional privilges. We list some possibilities below; this is not meant to be an exhaustive list.

Obviously, if a buffer overflow in Josh allows untrusted user Alice to obtain a root shell, that's a vulnerability.

If Alice can obtain read or write access to a file, or execute access to a program beyond what is granted to her by root, that's a vulnerability.

If Alice can obtain access to more of Bob's files than she should, that's a vulnerability.

If Bob can configure his files so that, as Alice runs Josh, she is tricked into giving Bob some subset of her privileges, that's a vulnerability.

If the system administrator, in going over the Josh journal files, is misled about the activities that took place on the system, or is tricked into giving up more privileges, that's a vulnerability.

Denial-of-service attacks may rise to the level of security vulnerabilities if they are serious: for example, if Bob can use Josh to disable Alice's login. (Attacks prevented by effective use of ulimits obviously don't count.)

In demonstrating an exploit, you may assume any reasonable configuration for the various files Josh uses to determine access; but, obviously, an exploit that derives from misconfiguration is not an exploit against Josh.