CS155: Computer Security **Spring 2003**

Project #2

Due: Thursday, May 15th, 2003.

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`.

This assignment specification is quite long, but much of it is intended to clarify points of confusion raised by previous years' students. Please be sure to read it carefully before beginning to code.

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`, mirrored on the class Website at `http://cs155/osh.html`). You might want to read the SANS III conference paper linked from the OSH home page, and mirrored on the class Website at `http://cs155/osh.sansIII.ps`.

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 add-on 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 We retain the Boxes distribution from `pp1`, as installed on the Linux machines in Sweet Hall. Since you will be doing Unix systems programming this time, we make available the `manpages-dev` package, omitted in the Boxes distribution, which contains man pages for Linux syscalls and C library routines. You can obtain this package from the CS 155 Website at

```
http://cs155/manpages-dev_1.48-2_all.deb
```

and install in in Boxes by saying, as root,

```
box:~# dpkg -i /path/to/manpages-dev_1.48-2_all.deb
```

(where you don't type the root prompt "`box:~#`" and where the actual path to the package replaces "`/path/to`").

Note that the filesystem image contains 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 in the Boxes distribution, and the additional information posted to the newsgroup for basic information on getting Boxes up and running.

Step One: Secure the Perimeter 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.

One thing to note: OSH currently searches the working directory for a program before searching the path for the program. This is dangerous (e.g., in the case when `/tmp` contains a malicious executable named "`ls`"). You should modify OSH so it executes programs in the working directory only if `$PATH` includes "`.`" or an empty component, or if a program is invoked as, e.g., "`./program`".

Another thing to note: Previous years' students have identified a buffer overflow in `substvars()`, a buffer overflow in `striparg()`, an array overflow in `pcmd()`, and other questionable code elsewhere. Be sure either to audit carefully or otherwise to take steps to mitigate the effects of vulnerabilities in OSH.

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. This path must be absolute in that it must begin with a "/". The path could, however, contain components that are "`..`" or symlinks.

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 `/bin/kill` invocation by Alice should run as root, even though Alice might have the Unix permissions to run `/bin/kill` as herself, user `alice`. This holds however Alice runs `/bin/kill`: via the absolute path, via a relative path, or via `$PATH`, but not if Alice runs a different program named `kill`, e.g., `/home/alice/local/bin/kill`.

Josh should allow users to run programs not listed in `josh_exec`; these programs will be run with as the user and with the user's ordinary permissions, precisely as if Josh had not been setuid.

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 must, of course, allow users to name programs via a relative path, via an absolute path, or through `$PATH`.

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

If `/etc/josh_exec` is writable by a user other than root, your Josh must abort on startup. If `josh_exec` is writable only by root, but not `root:root`, mode 600, Josh's behavior is up to you.

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, like `josh_exec`, should be installed `root:root`, mode 600. Like `josh_exec`, it must not be writable by anyone other than root; if it is, Josh must refuse to run. The `josh_access` file 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 multiple positive and negative entries are given on the same exact file, the last entry should win.

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.

To clarify, write access to a directory, in Unix, gives the right to add and remove files in that directory. Your Josh should never give this right when a user does not already possess it through her standard Unix permissions. A `+w` grant on a directory means that, for each file already in that directory, the user is allowed to write to that file, not that the user should be allowed to create new files in that directory. If a `+w` grant names a file that does not exist, Josh should decline to create it. (In other words, entries in `josh_access` have an effect only on regular files, not directories, and only on regular files that already exist.)

Note that, regardless of order in `josh_access`, more-specific entries always override less-specific entries. Thus the lines

```
bob:/home/user/bleh:+r
bob:/home/user:-rw
```

Mean that Bob should be granted read access to `/home/user/bleh`, even though the negative `/home/user` wildcard comes later in the access file.

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 ">").

To clarify, you need only provide these extra privileges for redirections, "<" and ">". You need not worry about providing privileges for filenames given as arguments on the command line. Even if `josh_access` includes the line "`bob:/etc/ssh/ssh_host_key:+r`", your Josh need not (and should not) grant any extra privileges to Bob when he executes "`cat /etc/ssh/ssh_host_key`".

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 (like the `/usr/sbin/vipw` command on BSD machines), 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. Note: This is not the only possible approach; you need not implement this particular approach.

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, if that variable is not defined, `/usr/bin/vi`.

The `edit` builtin should take a single argument, the name of the file to edit. The user must have both read and write access to the file in order to edit it.

The `edit` builtin, if invoked on a file not mentioned in `/etc/josh_access`, should allow the user to edit the file with her ordinary permissions.

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 pipeline run by a Josh user. Your log line should contain the user’s login name, the full command line, and one of “`OK`”, “`FAILED`”, or “`DENIED`”, separated by spaces.

`DENIED` applies only to redirects and the `edit` builtin. Suppose a user attempts to use one of these to access some file `foo`. Suppose further that the user doesn’t have sufficient Unix permissions for the operation independent of `josh_access`. In this case, if the file `foo` is mentioned in `josh_access` (either explicitly or implicitly through directory wildcards), but again without sufficient privilege, then Josh should log `DENIED`. (If the user doesn’t have Unix rights to `foo`, and `foo` isn’t mentioned in `josh_access` at all, Josh should log `FAILED`; if the user does have Unix rights to `foo`, Josh should log `OK`.)

`FAILED` also applies when a user attempts to execute some command for which `exec()` fails, e.g., a file that’s nonexistent or not executable (regardless of whether mentioned in `josh_exec`), or a program to which the user doesn’t have execute permission, but which is not mentioned in `josh_exec`.

`OK` applies when the program the user requests is `exec()`ed successfully, whether as root or as the user, and all redirections are allowed and succeed.

Note that you log an entire pipeline at once, but with only a single status code. Your log should let the administrator know the most serious error that occurred on the pipeline, if any. That is, if any of the commands in the pipeline is denied, the log should state `DENIED`; if none is denied, but any fails, the log should state `FAILED`; if all succeed, the log should state `OK`.

To clarify, every command issued by the user, whether or not it implicates Josh control files, should be logged. Commands separated by “`;`” should be logged separately; commands combined with “`|`” should be logged as a single pipeline. The logging unit is thus the argument passed to OSH’s `ppipe()`. The command portion of the log line should be exactly what is passed to `ppipe()`.

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 listed in a user's `.josh_access` file should be accessed as that user for both redirections and the edit builtin.

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`. (And exactly `~bob`, not, e.g., some subdirectory of it.) As with `/etc/josh_*`, Josh should refuse to act on the `~user/.josh_*` files if they are writable by someone other than user.

Root-granted positive privileges in `/etc/josh*` should always override user-granted ones. That is, if `/etc/josh_access` says that Alice has access to `~bob/foo`, Bob cannot deny that access in `~bob/.josh_access`.

Note that `~user/.josh_access` can only grant read access to files user can read, and write access to files user can write.

Restrictions Another team will audit your code. Out of consideration for them, please keep in mind the following restrictions.

You must write Josh in C, not C++ or another language. You may not make use of libraries not written by you (aside, of course, from OSH and `libc`).

You will not want to use the `getlogin()` function, since Expect doesn't correctly set it on `pty`s. Use `getpwuid()` or similar routines instead.

Simplifying Assumptions Because this is CS 155 and not CS 193u, we allow you to make some assumptions that limit the scope of the systems-programming component of the project.

You may choose to ignore all but the first 512 lines of all control files — `/etc/josh_exec`, `/etc/josh_access`, `~user/.josh_exec`, `~user/.josh_access`.

(You may not, however, assume that there always will be at most 512 lines.)

If you find it helpful, it is okay to read `/etc/josh_exec` and `/etc/josh_access` at startup and not reread them on every command execution.

For the editor, you may require that both read and write access are specified in `josh_access`. That is, you need not worry about a case where (for example) write access is granted to the user in `josh_access` and read access is available to the user through her ordinary Unix permissions.

You may assume that `/etc/josh_*` are sanely formatted. (You may not, however, make the same assumption about individual users' `~/.josh_*` files.)

You should make efforts to survive error conditions. Since OSH already terminates on many error conditions, you need not go out of your way to handle all new Josh-imposed error conditions without terminating. Be sure to fail-safe, however, even if you fail-stop.

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` (and `TA-README`, if you choose to include one).

You may also want to include a `README` file with details of your design; this file will be provided to the group that will audit your Josh, so be sure to include no identifying information in it.

Finally, you may want to include a `TA-README` file with comments about your experiences, or suggestions for improving the assignment; this file will not be provided to the group that will audit your code.

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 privileges. 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.