Programming Project # 2

cs155
Due 5/5/05, 11:59 pm
Elizabeth Stinson
(Some material from Priyank Patel)

Background – context

- Unix permissions model
  - Prof Mitchell will cover during OS security (next week – come to class!)
  - Many basic admin functions require root privileges
  - Problem: no way to assume SOME root privileges but not all
    - Coarse granularity

Problem

- How to provide root privileges for *certain tasks* to certain users
  - How to give bob root privileges so that he can kill arbitrary zombie processes but NOT edit /etc/passwd
- Who do we give the root password to? How do we log what people do while logged in as root? What happens if we don't give people the access they need? How to deal with changing the root password (distribution, revocation…)?

Solutions – I

- Operator Shell – Michael Neuman (LANL)
  - Not to be confused with the Old shell (OSH) which is essentially the starter code for `josh`
- Setuid root : so call seteuid(), setresuid(), … to elevate or depress current privilege level
- Limit access of commands and files to users who actually need that specific access
  - Granularity at a group level: consultants can do X, ALL can do Y; also can configure for a specific user
  - Specify a command matrix
- Maintain audit records (who touched what when)

Solutions – II

- Sudo: Jeff Nieusma
- "`Sudo` is a program designed to allow a sysadmin to give limited root privileges to users and log root activity."
- So you run `sudo` to execute a single command as root, e.g.; /etc/sudoers says who can do what
- Doesn't do access control for files
- Granularity of grant: able to execute a command or not; NOT able to control whether certain options with a command used

http://www.sudo.ws/sudo/man/sudo.html

Solutions – III (yours!)

- `josh`
- Start with OSH – Gunnar Ritter's Old Shell
  - Only 837 lines of code (C)
- To test this, you can put `josh` in boxes [use closedbox (on myths, vines)] – even create users whose default shell is `josh`
- When you're done: will have program access controls, file access controls, logging, a safe '`edit`' command, …

High bits

- Do a security audit of existing simple shell: make security bug (the type of which you exploited in pp1) fixes as necessary
- Add functionality to this shell:
  - Which would allow finer granularity of permissions granting (for execution of certain programs) vs. the Unix permissions model
  - Do likewise for certain file permissions
  - Add an editor
  - Add journaling – keep record of users' activities

More detailed – step 1 [Secure the Perimeter]

(You don't have to do these steps in this order)

- Buffer overflows (not just strcpy()'s) **etc.**
- OSH currently searches the working directory for a program before searching the PATH for the program;
  - So malicious user could put bad version of '`ls`' in `/tmp` dir so if another user exec's '`ls`' while in that dir, would be using corrupt code

OSH program execution

```
/* try cwd first */
execv(args[0], args);

/* try $PATH */
if ( errno == ENOENT )
  execvp( args[0], args );
```

Instead, try cwd only if '.' in $PATH

```
  PATH=/bin:.:/sbin
```

or empty path

```
  PATH=/bin/sbin::/usr/bin
```

More detailed – step 2 [Executables]

- Create a file: `/etc/josh_exec`
- Will contain entries of the form:
  `userid:progpath`
- `userid` : user's login name
- `progpath`: abs path to a program
- E.g. `alice:/bin/kill`
- Any program listed in `josh_exec` for a particular user should be executed as `root` when that user invokes it
- Of course just because `alice` has root privileges for `/bin/kill` doesn't mean that `alice` has root privileges for any executable named `kill`

A bit more on step 2 [Executables cont.]

`/etc/josh_exec` and `/etc/josh_access`

- Should be installed in mode 600;

```
if (writable by anyone besides root)
   then abort on startup;
else
   your choice
```

- Users should be able to run programs *not* listed in `josh_exec` (that those users would otherwise have access to) just *not to run as root*
- Users should, like normal, be able to run programs via an abs path, relative path, …

More detailed – step 3 [File access]

- An admin may need to read some files not readable by all users and/or write some files not writable by all users
- `/etc/josh_access` has entries of the type `userid:filepath:perms`
- `userid:` again the user's login
- `filepath:` again an abspath to a file
- `perms:` of the form [+-](r|w|rw)

More detailed – step 3 [File access cont.]

- `perms`
  + grants positive right
  - takes away *a previously granted right*
- Entries are cumulative
- Most specific wins (regardless of order)
- In equal specificity but contrary prescriptions (one is a `+r`, the other is a `-r`), most recent wins
- Can only take away rights granted by `josh`; that is, if a user has e.g. some read privileges in the OS, `josh` *should not take these away*

More detailed – step 3
[File access cont.]

`alice:/etc/motd:+w`

- gives `alice` permission to write motd file
- Everyone has read privileges on that file

`bob:/etc/ssh/ssh\_host\_key:+r`

`bob:/etc/ssh/ssh\_host\_key:+w`

- `bob` can read & write `ssh_host_key` file

More detailed – step 3 [File access cont.]

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

- If the filepath specifies a directory, the grant applies recursively to all files in that directory, except in the case that there is a more specific rule for such a file or subdir *canceling* that grant (remember, cancellations are only meaningful when overlapping previous grants exist)

More detailed – step 3 [File access cont.]

- Your access matrix should be used for redirection, too
  - To execute `gunzip < some_file`, the executing user must have read privileges on `some_file`
  - Likewise, to execute `ls -l > oth_file`, user must have write access to `oth_file`
- But doesn't need to be used when files are used as arguments to commands; e.g. `cat /etc/ssh_host_key`

Generally about the `josh_*` files

- Can read in on startup and thus not consult before every access decision
  - So if some superuser changed `josh_access` while `alice` was logged in, it's okay for `alice` to have the original privilege set that she had when she first logged in and got the `josh` shell

More detailed – step 4 [editor]

- Implement a shell built-in ‘edit’

```
edit /etc/rc.d/rc2.d/inetd_script
```

- Copy that file to `/tmp`
- Allow `alice` to modify the file in `/tmp`
- On exit from `vi`, copy back the file from `/tmp` to the original location
- Why doing this? Because there may be files for which a user doesn’t have native access (e.g. Unix file permissions) but for which josh grants read and write permissions to; we need a way to allow that user to edit such a file

More detailed – step 5 [logging]

- `josh` maintains 3 types of log events
  - OK : everything is fine
  - FAILED : exec call failed (file is non-executable or non-existent)
  - DENIED: tried to edit without sufficient privileges in Unix and `josh`
- Logging unit: single pipeline (the argument to ppipe())

Extra credit

- `alice` creates `josh_*` in home directory
- Grants access to other users for files which are readable and writable by `alice`
- Other users can access/execute with same privileges as `alice`

Generally

- Follow principle of least privilege
- Make reasonable checks on user input
- The original shell terminates on several errors; it's OK if `josh` does the same; be sure to fail safe even if you have to fail stop.
- Short functions good; promote program correctness (what you can see on one screen).
- Document your design in the `README`
- Start early.

Resources

- Michael Neuman's paper (Osh) linked to on site; is short, concrete and helpful
- W. Richard Stevens : Advanced Programming in the UNIX Environment – a great book, well worth your money
  - Good dets on chmod, execv*(), setuid, environment variables, …
- Setuid Demystified – linked to on site
- Old shell man page – quick tour through the starter shell (descr. functionality supported, …)

  http://jneitzel.sdf1.org/osh/man/osh.1.html
- Wheeler's "Secure Programming for Unix…"