UNIX Security: Security in Programming

Matt Bishop

Department of Computer Science
University of California at Davis
Davis, CA 95616-8562

phone (916) 752-8060
email bishop@cs.ucdavis.edu

Goal of Tutorial

Show you how to write programs which are to be run:

- by root (or some other user)
- are setuid or setgid to you (or root, or …)

and can't be tricked into doing what they are not intended to do

Why is This Hard?

Several reasons

- a "bug" here can endanger the system
- programs interact with system, environment, one another in sometimes unexpected ways
- assumptions which are true or irrelevant for regular programs aren't for these

What Do These Programs Involve?

- a change of privilege
  example: setuid programs
- an assumption of atomicity of some functions
  example: check of access permission and opening of a file
- a trust of environment
  example: programs which assume they are loaded as compiled

Security Policy

- What the program is allowed to do
  - Access a particular directory
- What the program is not allowed to do
  - Access any other files

Constraints imposed by the system administration, law, *etc.*

Example: Message Transfer Agent

Goal: accept and deliver mail

- Where to put it?
  - Any file allows it to be appended to /etc/passwd
  - Any program allows remote user to take arbitrary action
  - Must constrain delivery to known mailboxes, programs
- Forwarding Mail
  - How much information about system to include?
  - To which sites is it to be forwarded?
  - How to implement RFC 821's address rewriting requirements?

Design Principles

- Determine threats
  - to Confidentiality (best protected by end to end mechanism)
  - to Integrity (same comment)
  - to Availability (taking up disk space; mail-bombing)
  - delivery to unauthorized places (constrain where mail can go)
- Design with those threats in mind
- Include system constraints
  - Access to port 25 requires *root* privileges
  - Access to mailboxes requires extra privileges

Key Concepts

privilege running with rights other than those obtained by logging in; or running as superuser

protection domain
all objects to which the process has access, and the type of access the process has

Security Design Principles

Control design of all security-related programs

- principle of least privilege
- principle of fail-safe defaults
- principle of economy of mechanism
- principle of complete mediation
- principle of open design
- principle of separation of privilege
- principle of least common mechanism
- principle of psychological acceptability

Principle of Least Privilege

Process executes with only those privileges it needs

- what identity to assume
- what resources to access
- requires a privilege to be relinquished when no longer needed

"Need-to-know" rule

SMTP server runs as root to open the socket, but then reverts to *smtp* user (not root)

Principle of Fail-Safe Defaults

- Privileges by default are denied; they must be explicitly granted
- A failure should cause the original protection domain state to be restored
  - In both cases, if the program fails, the system is safe
  - MTA's spool directory should be read/write only by *smtp* user, not by anyone else (so the default is to deny access to queued mail)

Principle of Economy of Mechanism

- Same as KISS principle
- The simpler the design/mechanism, the easier it is to verify correctness and the fewer attributes or actions to go wrong
- Common problem points: interfaces, interaction with external entities
  - MTA split into multiple programs: server (to accept mail), client (to deliver mail)

Principle of Complete Mediation

Every access to every object must be checked

- UNIX OS violates this rule; checks only at open, not at reads
- Program should check data after each access for validity

On programs running as root, **nothing** is checked, so the program must do it

Principle of Open Design

Do not depend upon concealment of details or of security measures for security

Okay to use passwords, cryptographic keys, etc.

Security through obscurity:

- » adds some (easily overcome) protection
- » gives false assurance

Principle of Separation of Privilege

Grant access based upon multiple conditions

- root access depends on membership in group wheel as well as knowledge of the password
- access to operator conditioned on time, point of access, password, entry in authorization file
- use of a Kerberos ticket depends on time, authenticator

Principle of Least Common Mechanism

Minimize shared channels or resources

- » Avoid shared resources; some cannot be eliminated (common file system, CPU, memory, etc.)

Principle of Psychological Acceptability

Be kind to your users

- » Make the mechanism no more inconvenient than not using it
- » Make it acceptable to users
- » Make interfaces simple, intuitive

If mechanism too complex or cumbersome, users will try to evade it or will weaken it

Users and UIDs

Real UID:	UID of user running program
Effective UID:	UID of user with whose privileges the process runs
Login/Audit UID:	UID of user who originally logged in
Saved UID:	UID before last change by program

Example:

User *holly* logs in and executes file owned by user *matt*.

The resulting process has both a real and an effective UID of *holly*.

Setuid, Setgid Bits

```
% ls -lg a.out
-rwsr-sr-x matt sys 512 Nov  5 1988 a.out
```

example:

User *holly* executes this file. The process has:

Real UID: *holly*

Effective UID: *matt*

Login UID: *holly*

Saved UID: *matt*

Obtaining These UIDs

getuid() return real UID

geteuid() return effective UID

getauid() return audit (login) UID (varies)

On Solaris, must be root to run this

getlogin() return login (audit) UID

Warning: on some systems, getlogin returns the name of the user associated with the terminal connected to stdin, stdout, or stderr(which is *very* different than the above)

getsuid() returns saved UID (on some systems)

On others, your program must save this if you plan to refer to it later

Setting UIDs

setuid(uid) set UID
  if root, sets real, effective, saved; if not root, sets effective
setruid(uid) set real UID
seteuid(uid) set effective, saved UID
setauid(uid) set audit (login) UID (varies)
  On Solaris, must be root to run this
setlogin(uid) set login (audit) UID
setreuid(rid,eid) set real (rid), effective, saved UID (eid)

Groups and GIDs

Similar to users; group permissions apply to groups
Calls are analogous, with “g” replacing “u”.
getgid() return real UID
getegid() return effective UID
getsgid() returns saved UID (on some systems)
getgroups(int ngroups, int grouplist[ ])
  Get list of groups of current process; if ngroups too small, error is EINVAL

More Groups

setgid(gid) set GID

if root, sets real, effective, saved; if not root, sets effective

setrgid(uid) set real GID

setegid(uid) set effective, saved GID

setregid(rid,eid) set real (rid), effective, saved GID (eid)

setgroups(int ngroups, int grouplist[ ])

Set list of groups of current process; if ngroups too large, error is EINVAL

Getting User Names

```
#include <pwd.h>
struct passwd *getpwent(void);
```

```
   up = getpwuid(getuid());
   user_name = up->pw_name;
```

Returns first user with that UID

```
   up = getpwnam(user_name)
   user_uid = up->pw_uid
```

Returns first user with that name

Getting Group Names

```
#include <grp.h>
struct group *getgrent(void);

    gp = getgrgid(getgid());
    group_name = gp->gr_name;
    group_members = gp->gr_mem;
```

Returns first group with that GID

```
    gp = getgrnam(group_name)
    group_name = gp->gr_name;
    group_members = gp->gr_mem;
```

Returns first group with that name

Getting Login Names

```
char *getlogin(void)
char *cuserid(void)
```

Returns who is logged into the terminal associated with stdio, not the login name of the owner of the process

- » if stdin is associated with a terminal, get terminal name, look in /etc/utmp for user name
- » else if stdout is associated with a terminal ...
- » else if stderr is associated with a terminal ...
- » else return NULL

Attack

Goal: forge mail from Peter to Dorothy

Environment: Peter is logged into /dev/ttyha

Problem: mail program uses *getlogin* to get login name for return address

```
mail dorothy < letter > /dev/ttyha
```

No output, so Peter will see nothing; but letter comes to Dorothy from him!

Fixed on all 4.x BSD and System V systems that I know of

Starting Safe

Setuid program gives privileges for the life of the process, plus any descendants

Effect is same as if owner (not user) ran it

So … owner must dictate initial protection domain

Review: What Is Privilege

Here, it means program runs with rights not normally associated with user running it

Example: in *vi*, user cannot write to buffer storage area where file is to be put when user hangs up

so the process is given privileges (additional rights) to do it

Key Difference

setuid vs. a *root* (owner) process

- *root* process starts in root's environment
  - need not worry about change of environment
- setuid process starts in user's environment
  - must worry about change of environment

How Important?

In theory, major

- you can assume the trusted owner won't compromise system

In practise, relatively minor

- even root can make mistakes ...

Need to guard against stupid initial environments

Example: the Purdue Games Incident

Games very popular, owned as *root*

» Needed to be setuid to update high score files

Discovered that effective UID not reset when a subshell spawned

» So we could start a game which kept a high score file, and run a subshell – as *root*!

Ways to Fix The Problem

- Trust the Users
  - » Claim there is no problem as no user would ever do anything untoward in that case
  - » Overlooks nasty people who may gain access to your site
- Delete the Games
  - » Lots of support for this, but students had their own copies, and would have given one another setuid privileges ...
- Create a Restricted User
- Create a Restricted Group

Create a Restricted User

User *games* owns files in games directory, and no others

- » All game programs setuid to this user
- » High score files writable only by owner (*games*)

That user can delete games or score files but nothing else

Create a Restricted Group

Group *games* is GID of files in games directory, and no others

- » All games setgid to this group; may be owned by anyone
- » High score files writable by this group

That group can delete games or score files but nothing else

- » Further protection: make games unwriteable by group
- » Note high score files must be writeable by group and so can be deleted

Setuid vs. Setgid

If no need to log in, use group (not user)

- » Groups generally more restricted than owner

If group compromised, usually much less dangerous

- » Due to usual system configuration; not inherent

Application of privilege of least principle

Example: The *crash*(8) Attack

problem: *crash* is setgid to *kmem,* which is the group of the memory device files

If you get a subshell, the effective group id is not reset

```
host% crash
dumpfile=/dev/mem, namelist=/vmunix, outfile=stdout
> !sh
```

and you can now read */dev/mem* (or worse, write it)

Fixes: • turn off setgid bit on crash
• turn off all group permissions on memory devices

Example: The *ps*(1) Attack

Goal: read any location in kernel memory

ps accesses process table by:

- » opening symbol table in */vmunix*
- » looking up location of variable _proc

ps setgid to group kmem

User can specify where *vmunix* file is

So supply your own */vmunix* and read any file that group kmem can read ...

Design Tip: Use of Setgid

- A setgid program can be just as dangerous as a setuid one
- A non-privileged program run by a privileged user can be as dangerous as a setuid program
- Protection domain includes user and group identity

fork, *exec*, and UIDs and GIDs

UID and GID are preserved across execs

setuid changes EUID and saved UID, setgid changes EGID and saved GID; these stay with process when interpreter overlaid

UID, GID preserved across *fork*(2)

all are unchanged; new process has those of the old parent process

Programming Tip: Spawning Subprocesses

Reset UID, GID after fork to the real UID, GID

… unless there is a very good reason not to

Environment

process/system interaction

- » via system calls

process/process interaction

- » via shared files, signals, etc.

process/descendant interaction

- » via forking, pipes, shared resources, etc.

Note environment variables fall under third class

Starting Example

vi *file*

… edit it, then hang up without saving it …

- vi invokes expreserve, which saves buffer in protected area
  ... which is inaccessible to ordinary users, including editor of the file
- expreserve invokes mail to send letter to user

Where Is the Privilege?

vi is not setuid to *root*

» you don't need that to edit your files

expreserve is setuid to *root*

» the buffer is saved in a protected area so expreserve needs enough privileges to copy the file there

mail is run by expreserve

» so unless reset, it runs with *root* privileges

The First Attack

```
$ cat > ./mail
#! /bin/sh
cp /bin/sh /usr/attack/.sh
chmod 4755 /usr/attack/.sh
^D
$ PATH=.:$PATH
$ export PATH
```

… and then run vi and hang up.

Design Tip: The PATH Environment Variable

Don't trust the setting of the user's **PATH** variable

- » if your program will run any system commands, either give the full path name or reset this variable explicitly
- » This by itself is not enough, however ...

So `vi` Fixed it …

Instead of resetting **PATH,** change

```
system("mail user")
```

to

```
system("/bin/mail user")
```

But … still uses Bourne shell …

The Second Attack

Bourne shell determines whitespace with **IFS**

Using same program as before, but called *m*, do:

```
% IFS="/binal\t\n "; export IFS
% PATH=.:$PATH; export PATH
```

… and then run vi and hang up.

Design Tip: The IFS Environment Variable

Don't trust the setting of the user's **IFS** variable

- » if your program will run any system commands, reset this variable explicitly
- » must still deal with **PATH**

Fixing This

Fix given in most books is:

```
system("IFS='\n\t ';PATH=/bin:/usr/bin;\
         export IFS PATH;command");
```

This sets **IFS**, **PATH**; you may want to fix more

<u>WRONG</u>

How to Break This

Before invoking your program *plugh*, I do:

```
% IFS="I$IFS"
% PATH=".:$PATH"
% plugh
```

Now your IFS is unchanged since the Bourne shell interprets the I in `IFS='\n\t '` as a blank, and reads the first part as `FS='\n\t '`

Programming Tip: Explicit Environment Variables

Look for any code using environment variables:

```
main(argc, argv, envp)
extern char **environ
getenv("variable")
putenv("variable")
```

The only time you should use them is when they do not affect the security of the program

Programming Tip: More on Environment Variables

Can add them directly to environment, so multiple instances of a variable may occur:

```
PATH=/bin:/usr/bin:/usr/etc
TZ=PST8PST
SHELL=/bin/sh
PATH=.:/bin:/usr/bin
```

Now which PATH is used for the search path?

- Answer varies but it is usually the second

If PATH is deleted or replaced, which one is affected?

- Usually the first ...

Programming Tip: Implicit Environment Variables

These functions call the shell or use **PATH**:

- *system*(3), *popen*(3)
  - Call the Bourne shell
- *execlp*(3), *execvp*(3)
  - These use **PATH**
- any *exec* derivative
  - These may implicitly pass the environment along

Programming Tip: Controlling Environment Variables

Use *execve*(2)

You then reset what parts of the environment you want:

```
envp[0] = NULL;
if (execve(path_name, argv, envp) < 0) ...
```

Note: may have to set TZ on System V based systems

Use *msystem*(3) or *mpopen*(3)

These provide interfaces to *execve*; discussed later

Never use *system*(3) or *popen*(3)

unless you clean out your own environment first

Analysis of These Problems

Programs run with more privileges but in an environment set up by a user with fewer privileges

This means programs trust and (implicitly or explicitly) use this environment

Dynamic Loading and Environment

- General assumption: programs loaded as written
  - this means parts of it don't change once it is compiled
- Dynamic loading has the opposite intent
  - load the most current versions of the libraries, or allow users to create their own versions of the libraries

How Dynamic Loading Works

- On execution, library functions not loaded
  - Instead, a stub is put in its place
- When library function called, stub loads it
  - Stub figures out where to look, pulls file out of library archive, puts it into memory
- Execution then jumps to the loaded function

The Problem

Where is this new routine obtained from? Possibly an environment variable …

- On Suns: check libraries in directories named in the variables **LD_LIBRARY_PATH**, **LD_PRELOAD**; those directories are searched in order, just like **PATH**
- Other systems have similar mechanism (**ELF_** variables, *etc*.)

This puts execution of parts of a setuid program under user control

- ... as the user controls what is loaded and run

Attack: the Setup

- Find a setuid program that uses dynamic loading (here, we'll use /bin/login, which dynamically loads the routine *fgets* to read the login name)
- Build a dynamic library with your own version of *fgets.o*:

```
fgets(char *buf, int n, FILE *fp)
{
  execl("/bin/sh", "-sh", 0);
}
```

- Put it into a library *libme.so* in current directory

The Attack

Execute the following

```
% LD_PRELOAD=.:$LD_PRELOAD
% /bin/login
#
```

You now have a login shell with privileges of the owner of login, namely root

The Obvious Fix

Problem: Dynamic loading allows an unprivileged user to ater a privileged process by controlling what is loaded

Idea: Disallow this control by having setuid programs ignore environment variables

- Here, they would dynamically load libraries from a preset set of directories only

Reasoning: Users can control what is dynamically loaded on their programs, but not on anyone else’s, since everything you do is executed under your UID or is setuid to someone else …

Close, But No Cigar

Flaw in the Analysis: Suppose a setuid program runs a non-setuid program?

- Login does this (it spawns the login shell, or some other designated program, which is not setuid)

Result: The non-secure variable is ignored by the setuid program and is propagated to the non-setuid program

But the non-setuid program is not running with the privileges of the user; the setuid program can change these, especially if run by root

The *sync* Account

How login works:

- By default, *login* clears current environment
- –p option preserves current environment

Can use any account for what follows, but need to complete login; as *sync* has no passowrd on most systems, an obvious candidate

User is UID 1 (daemon); login shell is /bin/sync

- dynamically loads the system call sync()

The Attack

- Execute the following

```
% LD_PRELOAD=.:$LD_PRELOAD
% /bin/login -p sync
%
```

You now have a shell running with *daemon* privileges

What Happens?

- login ignores LD_PRELOAD and works as expected since it is setuid
- /bin/sync uses LD_PRELOAD since it is not setuid, even though it executes as *sync*
  - Effect: current user can control execution of another user’s program

Another Example: Loadmodule

```
$ PATH=.:$PATH
$ cat > /bin/ld
#! /bin/sh
sh
^D
$ cp /usr/openwin/loadmodule/evqload
  evqload
$ cp /usr/kvm/sys/sun4m/OBJ/sd.o sd.o
$ loadmodule evqload sd.o
#
```


What Are the Causes

First one we’ve seen

- program not specified fully

  a full path name not given; probably **IFS** not protected either

This one’s been implicit, but now it’s explicit

- environment not reset to trusted state

  should turn off dynamic loading as *loadmodule* is setuid to *root;* dynamic loading involves a loading program which is trusted, so make sure the assumption of trust is incorrect (ie, use a fake program)

Programming Tip: Don't Dynamically Load

Most loaders on such systems have an option which specifies static binding

On Suns, it's –Bstatic; with *gcc*, it’s –static

Use it on anything that will be run setuid or setgid

Design Tip: Know What You Trust

Know where your trust is!

- if dynamic loading is a possibility, and you can disable it, do so
- if you can eliminate dependence on environment, or check assumptions about the environment, do so
- if you can't, warn the installer and/or user

Moral: identify trust points in design *and* implementation

To Sum Up

Class of flaws is “improper change”

Violates design principles (least privilege, least common mechanism, fail-safe defaults)

Whenever you change privileges (such as with a setuid program), you cannot trust the old, unprivileged environment

Best to avoid any such programs if you can

More on this later

A Second Point of View

General class is improper choice of initial protection domain

... as users can reset protection domain at will

Fix: force a specific protection domain into the program

Minimizes trust in environment

Validation and Verification

Distrust anything the user provides

ps: if using */vmunix*, namelist is (probably) okay; if using something else, namelist is (probably) not okay

» Why? Because first assumed writeable only by trusted user (who can read memory (root; this should be checked both at /vmunix and at /dev/kmem). Assumption for other users is likely to be wrong at both points.

» Effectively, above fix allows user to supply alternate namelist only if user could read memory file anyway

The (Apocryphal?) Login Bug

Declaration in *login.c* is:

```
char name[80], passwd[80], hash[13];
```

- user types name
- hash loaded with corresponding password hash
- user types password, hash for that password

password and hash validate; she's in!

The Fingerd Bug

input stored in a character array allocated as

```
char buf[256]
```

- *fingerd* uses *gets* to read *buf*
- enter 289 chars, not 256

This overflows *buf* , overwriting return address

The *syslogd* Bug

- *syslogd* reads message from a socket
  - does *not* use *gets*, so no overflow there
- message formatted with PID,date, etc.
  - uses *sprintf* with an array *line2* allocated at 2048 characters

Array for *sprintf* can overflow just as in previous slide

The Problem

In all cases, string put into array without being checked for overflow

- if *passwd* not overflowed, *hash* not altered and correct password validated
- if *buf* not overflowed, stack uncorrupted and return made to *main*
- if *line2* not overflowed, stack uncorrupted and return made to caller

Design Tip: Buffer Overflow

Assume input may overflow an input buffer

Design to prevent overflow

In general, don't trust input to be of the right form or length

Called an improper validation condition

Programming Tip: Handing Arrays

Use a function that respects buffer bounds

Avoid these:

gets *strcpy* *strcat* *sprintf*

Use these instead:

fgets *strncpy* *strncat*

(no real good replacement for *sprintf*; *snprintf* on some systems)

To find good (bad) functions, look for those which handle arrays and do not check length

» checking for termination character is *not* enough

Invalid Input

Get IP address 555.1212.555.1212; want host name

Use *gethostby addr*, which uses Directory Name Server

Response p used as:

```
sprintf(cmd, “echo %s | mail bishop”, p);
if (msystem(cmd) != BAD) ...
```

Assumption: *gethostbyaddr* is reliable, meaning DNS is reliable

» but it's not under our control

The Faulty DNS

Say host name resolves to

```
info.mabell.com; rm -rf *
```

Command executed is

```
echo info.mabell.com; rm -rf * | mail bishop
```

Attacker has executed command on my system

User Specifying Arbitrary Input

Need to check any string being used as a command and originating elsewhere

- Good example: when user supplies value for environmental variable DISPLAY

Say string has any metacharacter meaningful to shell

- Examples: | ^ & ; ` < >

If user gives a recipient for mail as

```
bishop | cp /bin/sh .sh; chmod 4755 .sh
```

then using this as an address to mail command gives a setuid to (process EUID) shell

- Bug in Version 7 UUCP, some versions of *sendmail*, some versions of Web browsers

Programming Tip: Unreliable Information

Whenever data is read from a source the process (or a trusted user) does not control, *always* perform sanity checking

- » for buffers, check length of data
- » for numbers, check magnitude, sign
- » for network infrastructure data, check validity as allowed by the relevant RFCs; in DNS example, ; * ‘ ‘ all illegal characters in name

Example of improper verification of data

Other Sources

Not just data; also information from system

- assuming ownership implies other things, such as permission
  - » okay if the owner had to copy file or affirmatively initiate the action; not okay otherwise
- assuming a name is tightly bound to an object
  - » for file descriptors, this is true
  - » for hard links, this is false
  - » for symbolic links, this is *really* false

Ownership and Permission

on one system, *at* queued requests; *atrun* executed them

- *at* not setuid; instead, *at* directory world writable
- *atrun* setuid, so it could run job as right user

atrun took owner of queue file as the name of the user who made the request, and executed with that user's permission

Bad assumption!

Users can write to files owned by others; eg. mailboxes

The *At* Attack

- Mail set of shell commands to *root*

  More generally, put commands into a file owned by another
- Link file into at directory with correct name

  As mail and at directory on same device, real easy
- *atrun* will execute the mail file commands

  and as *root* owns the mailbox, commands execute with *root* privileges

What Happened

Problem: *atrun*'s validation technique flawed

as anyone can create or link a file into the *at* directory, can't trust that *at* put all files (and hence all jobs) there

Solution: make *at* setgid and *at* directory group writable, but not world writable

then *at* must be used to do the queueing and the owner stays associated with the command file

Another Failure to Check

- *Lpr* spool files are identified by a 3-digit unique number assigned sequentially (essentially, the job number)
- *Lpr* was setuid to *root* and opened the spool files for writing *without* checking to see if the spool file already existed
- *Lpr* allowed queueing of symbolic link as well as regular file

Overwriting Any File

- Create a small file *x* containing the password file
  for best results, make the *root* password field empty
- Start printing a big file using a symbolic link
- Queue the password file, again using a symbolic link:

```
lpr -s /etc/passwd
```

- Print 999 files
  this must be done before the big file finishes printing
- Now print *x*

```
lpr x
```

password file overwritten

Why?

Lpr writes the contents of *x* into the spool file that is a symbolic link to /etc/passwd; and writing to a symbolic link alters the file that the link points to

Lpr can alter any file as it is setuid to *root;* /etc/passwd is modified

Assumptions:

- Never be more than 999 files queued at a single time
- *Lpr* will never overwrite anything not in the spool directory

Fixes

Fixes:

- Make *lpr* setgid to daemon, *etc*.
- Check that the spool file being written to does not exist; if it does, stop, or delete it and then write

Note:

- Increasing the number from 3 digits to more will make this attack less likely to work (*i.e.*, more difficult to execute) but will not block it

Opening Files

Flags to modify *open* system call:

O_APPEND append data to file when writing

O_CREAT create file if it does not exist

ignored if file exists

O_CREAT|O_EXCL create file if it does not exist

give E_EXIST error if it does exist; symbolic links *not* followed

On creation, owner and group set as follows:

- owner set to EUID of creating process
- group set to EGID of creating process

  some systems: if directory is setgid, file gets directory's group

Problem

File can change between access check and printing

Fix #1: modify *lpd* to check access mode of file being printed relative to user who queued request

Fix #2: Make *lpr* setgid to *daemon*

- requires *daemon* to be able to read any file you want to print
- can still print any file *daemon* can read, even if you can't

Many vendors do this (System V variants)

Design Tip: Directory and File Permissions

If storing information, do not do so in a file or directory that an untrusted user can write to

sufficient to control access if you do so completely

In *at* case:

- information here is owner of file
- user can write to directory, so access not completely controlled

In *lpd* case:

- user can effectively write to queued file, so access not completely controlled

Design Tip: Verification

Think through very carefully how you check access and data

Never trust the user to give you correct information or to abide by your program's expectations

Do not trust data from non-secure servers in the network (especially the DNS!)

Sendmail Hole

Goal: read any file on the system

- *sendmail* ran setuid to *root*
- –C option used to test (and debug) *sendmail.cf* file
- excellent error diagnostics, giving line and pointer to the error

Sendmail Attack

```
sendmail -C protected_file
```

Output is:

```
when in the course of human events
---error: bad format
it becomes necessary for a people to declare
---error: bad format
```

so delete every other line!

One Partial Fix

use *access*(2) system call:

```
access(config_file, R_OK)
```

if < 0, real user can't read file; so *sendmail* shouldn't read it on his/her behalf

Warning: this solution is probably flawed!

The hole exists only under very specific conditions (more on this later) and is much smaller, but still exists

Programming Tip: Files and Directories

When checking for access, check for file type also

- if file is symbolic link, check access on each component in the links until you reach the end

When checking for ability to write, check ancestor directories also

more on this later

When checking for ability to read or write, check for real UID's (GID’s) access, not effective UID's (GID’s) access

Co-operating Processes

4.2 BSD line printer spooling system:

- *Lpr* queued files, *lpd* printed them
- *Lpr* was setuid to *root* and spool directory not world-writable
- *Lpr* allowed queueing of symbolic link as well as regular file

The Lpr Attack

Relied on assumption *lpd* made about identity of requester

- Specific assumption was that *lpr* checked it and file could not be changed afterwards

```
% ln -s x y
% lpr some_huge_file
% lpr -s x
% rm -f y
% ln -s y some_unreadable_file
```

and out comes *some_unreadable_file* …

Specific Problem

lpr checks file attributes and permissions and assumes they won't change

- as file in protected directory, seems reasonable

using a symbolic link protects the link and not the object (file)

- so we change the referent after the check (by *lpr*) and before the use (by *lpd*)

Similar Problem in *sendmail*

Previous fix is roughly

```
if (access(config_file, R_OK) < 0) error
fp = fopen(config_file, "r");
```

But may not be good enough ...

Attack: change files between *access* and *fopen*

Why This Can Work

Want to check permissions and open as a single operation; cannot be done unless check is for effective UID/GID

- checking for access based on real UID/GID requires *access*(2) followed by *open*(2), and there is a window of vulnerability between the two; no guarantee that the object opened is the same as the one checked

Example of class of improper indivisibility flaws

Very Old Bug

From UNIX Version 7:

- no *mkdir*(2) system call to create a directory
- used a 2 step process:
  - *mknod*(2) to make directory
  - *chown*(2) to change owner from *root* to user

Flaw

To wind up owning the password file:

- make . writable
- execute *mkdir*
  - after *mknod,* but before *chown*:
    - » delete directory made with *mknod*
    - » make a link to */etc/passwd*

Result: user owns */etc/passwd*

How To Fix This

In Version 7, *mknod*(2) had to be executed by *root*

- must *mknod*, *chown* in one operation
- UNIX V7 doesn't have such a primitive
- So add it: *mkdir*(2) primitive
  - that's why it was added in BSD

Design Tip: Atomicity

When designing, think of what operations must be atomic

- use atomic primitives when possible
- when not, warn installers (and users) and minimize window of vulnerability

Programming Tip: Atomicity

Favor system calls over library functions
- the former are atomic, the latter usually not

Don't be afraid to fork, reset UID, and use pipes
- idea is the unprivileged process does the I/O and other risky operations; more on this later

Another Race Condition: Shell Scripts

How executed on most systems:

Kernel picks out interpreter
- first line of script is #! /bin/sh

Kernel starts interpreter with setuid bits applied

Kernel gives interpreter the script as argument

Window of Vulnerability

Between second and third step, replace script with file of your choosing

- cp /bin/sh .sh; chmod 4755 .sh

You've now compromised the user

Design Tip: Setuid Scripts

In general, don't

- too easy to create a security hole

If you must, provide a wrapper which is setuid and which will honor the setuid bits on the script

- then simply exec the interpreter yourself, open the script, and use *fstat* to check the bits

Logging from a Privileged Program

Problem: privileged program wants to write to a file owned by the real (not effective) UID

- may have to create it

Why? Allows logging (very useful for system facilities)

The *Xterm* Logging Facility

Xterm must run setuid to *root* to access device files

- else, others can interfere with it; also needs to update protected files

Xterm also want to let user log session (input and output)

Xterm and Logfiles

Xterm did not check access protections on log files

```
$ cat >! /tmp/imin
newroot::0:0:Watch this, turkeys!:/:/bin/csh
^D
$ xterm -l -lf /etc/passwd -e cat /tmp/imin
```

… and now you can *su* to newroot

Saw this before (with *sendmail*)

Moral: problems recur

First Iteration

New sequence to replace the old one (X11R5?)

```
if (access(log_file, R_OK) < 0) …
fd = creat(log_file, 2, 0644);
```

if file doesn't exist

```
chown(log_file, bishop, sys);
fd = open(log_file, 2);
```

if file exists

Better: checks access.

But It's Not Over

Notice window between *access* and *chown*, or *creat* and *chown*

- » Attacker uses symbolic link for log file
- » Process passes *access*
- » Before *chown*, make link point to */etc/passwd*
- » Proceed as in attack #1

Next Iteration

Do *open*(*creat*) first, then *access* check and *chown*

```
if ((fd = open(file, O_WRONLY)) > -1){
     if (faccess(fd, W_OK) < 0 ||
            (fchown(fd, uid, gid) < 0)){
          close file...
```

Must use *faccess* and *fchown* for this!

many systems do not have them

Will not work if *fchown* or *faccess* is replaced by *chown* or *access*

Better Solution

Eliminate the problem by having the check and open done atomically (by the kernel)

Idea is to make real UID the effective one

- » create pipe
- » fork
- » setuid of child to real UID (real UID now = effective UID)
- » child opens the file for writing, and copies from the pipe to the file
- » parent logs by writing to pipe to child, not directly to file

Design Tip: Closing Windows of Vulnerability

These occur when:

- privileged process acts on information that changes between validation and use
- the checking and use is not atomic

To prevent this hole, ensure checking and passing of information is atomic

simulated with *faccess* and *fchown*

simulated with pipes; OS does the checking

Key Point

File descriptors are not synonyms for file names!

File (data + inode information) is object

File descriptor is variable containing object

- Bound once, at file descriptor creation; hence, once open, a file's name being changed doesn't affect what the descriptor refers to

File name is pointer to object, with loose binding

- Name rebound at every reference

Precise Problem

More precisely, in something like

```
if (access("xyz", R_OK) == 0)
        fp = fopen("xyz", "r");
```

if user can change binding of *xyz* between the check (*access*) and the use (*open*), the check becomes irrelevant

A Classic Race Condition

Problem:

- access control check done on object bound to name
- open done on object bound to name

  no assurance this binding has not changed!!!

Solution: use file descriptors whenever possible, as once object is bound to file descriptor the binding does not change.

Warning:

names and file descriptors don't mix!!!

Another Instance

Warning:

names and file descriptors don't mix!!!

```
fp = fopen("xyz", "r");
if (access("xyz", R_OK) == 0)
   ... use fp ...
```

still has the race condition, as opening an object binds the descriptor to the object, ***not*** to the name

access(2) Safe Usage

Use `faccess(int fd, int mode)` if your system has it

```
fp = fopen("xyz", "r")
if (faccess(fileno(fp), R_OK) < 0)
    fclose(fp)
```

Safe if *path* is a regular file/directory or device, and it and all ancestor directories are unwritable by any untrusted user

If not safe, open pipe, fork, reset effective UID, access through the subprocess

Programming Tip: Using *access*(2)

Just because you can do it doesn't mean you should!

- Don't rely on access in general
  - you can in the specific case where no untrusted user can write to a directory or any of its ancestor directories
  - If directory or any ancestor is symbolic link, check link, then repeat *full* check on referent
- Use subprocesses freely

File Descriptors and Subprocesses

- These are not closed across fork or exec
- Threat is when privileged parent opens sensitive file and then spawns a subshell

Example of Problem

```
main()
{
        int fd;
        fd = open(priv_file, 0); dup(9, fd);
        (void) msystem("/bin/sh");
}
```

Running this and typing

```
% cat <&9
```

prints the contents of *priv_file*

Design Tip: Open Files

Access privileges checked on *open* or *creat* only

not checked on read, write, *etc*.

This is how pipes work; also useful for log files

- » open rotected log file as *root*
- » drop privileges to user
- » can still log data in protected file

Programming Tip: Closing Across *exec*

Close sensitive files across execs:

```
fcntl(9, F_SETFD, 1)
```

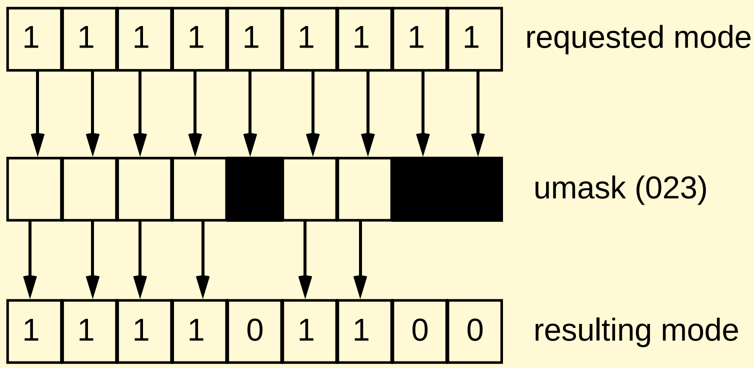
on System V and 4.xBSD; or

```
ioctl(9, FIOCLEX, NULL)
```

on 4.xBSD

File Creation Permissions

Used to clobber permission bits when creating files:



Umask Is Inherited

If not set to a safe state (preventing reading or writing for *world*), the *exec'ed* program may create world-readable/writable core files, or world-writable *root*-owned files and/or directories.

May enable attacks (*at* hole) or allow confidential information to be seen (in a core dump)

Programming Tip: umask

Reset this to a safe state

- » definitely turn off world write permission; turning off group write is usually good too
- » can turn off read permission for those folks; definitely do so if there is sensitive information, like passwords, in memory

How?

```
umask(022)
```

turns off group, world write

Programming Tip: For *root*

By default, *root* has umask of 0

Daemons start up with logs created mode 666 (a=rw) so system manager can configure permissions

So, in */etc/rc.whatever*, say

```
umask 022
```

to make logs mode 644 (u=rw,go=r)

A General Observation

There's more to an environment than environment variables

UIDs	root directory of process
GIDs	file system paths of referenced files
umask	network information
open file descriptors	process name

Essentially, environment is the protection state of the system plus anything that affects that state

Design Principle: KISS

Interaction with environment too complex:

- need to handle environment variables
- need to worry about loaded routines

Goal: minimize interactions

make the program as self-contained as possible

Example of the *principle of least common mechanism*

Setuid Shell Scripts

Very dangerous even when done with wrappers

Shells are too powerful and interaction with environment can produce unexpected results

example: if arg 0 begins with '–' it's a login (interactive) shell

And on Some Systems

```
% ls -l /etc/reboot
-rwsr-xr-x 1 root   17 Jul 1992 /etc/reboot
% ln /etc/reboot /tmp/-x
% cd /tmp
% -x
#
```

Design Tip: Assumptions

Don't assume user cannot control the name of the program

Here, assuming user can't put a "–" in char 0 of arg 0; also assuming login shell must be interactive

Don't assume user will enter a valid part of a command

Here, just a name and not a name plus more

Yopu saw this one earlier, too

Programming Tip: Names

Don't base user's ability to control actions of program on program name

- Okay to have name determine what program does
- Not okay to allow user to alter program's actions during run based solely on name

Example of Principle of Separation of Privilege

- base such permission on more than one check, such as name and password

That Old *su* Bug (Apocryphal?)

If *su* could not open password file, assumed catastrophic problem and gave you root to let you fix system

Attack: open 19 files, then *exec su root*

At most 19 open files per process, so …

Note: Possibly apocryphal; a non-standard Version 6 UNIX system, if true

Design Tip: Error Recovery

With privileged programs, it's very simple:

DON'T

Why? Because assumptions made to recover may not be right

In above, error was to assume open fails only because password file gone

Example of Principle of Fail-Safe Defaults

Design Tip: When to Recover

Track what can cause an error as you write the program

Ask "What should be done if this does go wrong?"

If you can't handle all cases, or determine precisely why the error occurred, or make assumptions that can't be verified, **STOP**

Programming Tip: Errno

```
#include <errno.h>
extern int errno;
```

Precise cause of failure often put in here

- for *su*, example sets *errno* to **EMFILE**
- for su, no password file sets *errno* to **ENOENT**

Warning: not automatically cleared, so program must clear it (set it to ENONE or 0)

Programming Tip: General Use of System Calls

Never assume a system call will succeed!!!

- If the success of a system call (such as *read*) is crucial to the program's success or failure, check the return code to be sure it is not -1.
- This applies to library calls, functions defined within the program, and *everything*

.

Programming Tips: System and Library Calls

Next slides give tips about using some functions not discussed earlier

Format:

- include files
- call
- exact meaning/effect

Non-network calls only here!

access(2)

```
int access(char *path, int mode)
```

returns 0 if *mode* access to path allowed to real UID/GID

returns –1 if not

mode: 4 (read), 2 (write), 1 (execute), 0 (exist)

Warning: dangerous call, unless used carefully; see earlier discussion

- » file must be writeable only by trusted users
- » all ancestor directories must be writeable only by trusted users
- » if any component is a symbolic link, iterate on referent

chmod(2)

```
int chmod(char *path, int mode)
int fchmod(int fd, int mode)
```

- » changes mode of file to mode
- » if file is open, use *fchmod* not *chmod*
- » umask ignored

- Warning: if EUID not root, this may turn off setuid, setgid bits
- Warning: if sticky bit set on directory, only root or owner of file can delete or rename file
- Warning: follows symbolic links

chown(2)

```
int chown(char *path, int mode)
int fchown(int fd, int uid, int gid)
```

changes UID, GID as specified; set either to –1 to leave alone

if file is open, use *fchown* not *chown*

- Warning: this may turn off setuid, setgid bits
- Warning: changes owner of symbolic link, not what link points to

chroot(2)

```
chroot(char *dirname)
```

Changes the process' notion of root directory "/" to be *dirname*

Problems:

- » can be used to acquire superuser status
- » may not work right if directory tree set up badly

Warning: Don't do this to restrict superuser

- » superuser can issue *mknod* system call to build device corresponding to kernel memory
- » superuser can then edit root directory field of process in process table

chroot Problem #1

Goal: switch to *root*

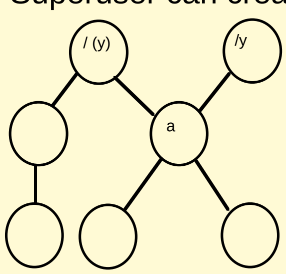
```
% mkdir /etc
% echo 'root::0:0:0:me:/:/bin/sh' > /etc/passwd
% exec su root
```

As root directory is inherited across *forks* and passed along *exec*s, *su* uses new */etc/passwd*; user is *root*

chroot Problem #2

Goal: break out of restricted subtree

Superuser can create (hard) link to directories



Here, "a" was initially subdirectory of "/x". Superuser linked it into the tree rooted at "/y".

User logs in and is chrooted to have "/y" as her root. She does:

```
cd /a/..
```

and her current working directory is "/x".

creat(2)

Manual says *creat* can be used for locking, as you can't *creat* an existing file:

User A:

```
if ((fd = creat("/tmp/x",0))<0)
    locked out
```

User B:

```
if ((fd = creat("/tmp/x",0))<0)
    locked out
```

Idea is user B's fails as B cannot *creat* a file A has created

The Right Way to Do File Locking

Use *link*(2), which **always** prevents creation of an existing link:

User A:

```
if (link("/etc/rc","/tmp/x")<0)
    locked out
```

User B:

```
if (link("/etc/rc","/tmp/x")<0)
    locked out
```

If /etc and /tmp are on the same file system, B's link fails if A's succeeds **even if** B is *root*

Other Ways to Lock Things

```
int flock(int fd, int operation)
```

returns 0 if operation succeeds, –1 if not

Operation is any of:

1 (shared)	4 (non-blocking)
2 (exclusive)	8 (unlock)

Warning: advisory lock only; processes may ignore it!

exec(2)

new process inherits:

real UID, GID	seconday group list
working, root dir	umask
blocked signals	environment variables

- effective, saved UID, GID (unless setuid/setgid file)
- open file descriptors (unless marked closed on *exec*)

fcntl(2)

```
#include <fcntl.h>
int fcntl(int fd, F_SETFD, int closeit)
```

if closeit is 1, close fd on exec

if closeit is 0, leave file open on exec

use `fcntl(int fd, F_GETFD, 0)` to see status

fork(2)

```
int fork(void)
```

inherits a copy of everything from parent

Note: private copy of open file descriptors, environment variables

- so closing them doesn't affect parent

fsync(2)

```
int fsync(int fd)
```

Synchronizes disk copy with any in-core modifications

Useful when files need to be updated on disk to keep consistent with in-core copies

getpgid(2)

```
int getpgrp(int pid)
```

Returns group number of process (in effect)

Any process in this group can signal this process

Useful for controlling who can suspend or terminate process as well as read or write controlling terminal

TIOCGPGRP, TIOCSPGRP

```
int ioctl(int tty_fd, TIOC?PGRP, int pid)
```

get/set process group number

if process not in process group tries to read controlling tty, gets a **SIGTTIN**

if process not in process group tries to write controlling tty, and **LTOSTOP** bit set in tty local modes, and process not in *vfork*(2), gets a **SIGTTOU**

Control Terminal

Always named */dev/tty;* refers to terminal from which process is run

How to change:

- if no associated control terminal, first one opened becomes control terminal
- disassociate by

```
ioctl(tty_fd, TIOCNOTTY, 0);
```

- to pretend a char was typed at tty, use

```
ioctl(tty_fd, TIOCSTI, &ch)
```

Attack

Goal: to run *date*(1) as though typed at console

```
char *str = "date\n";
ioctl(tty_fd, TIOCNOTTY, 0);
fd = open("/dev/console", O_WRONLY)
while(*str)
   ioctl(fd, TIOCSTI, str);
```

Now any process in the process group which is reading the terminal will take date as input

Fix

Make all terminal devices unwritable by other

Make all terminal devices in group tty

Make all programs which write to terminal setgid to tty

- Such as talk, write, etc.

Then *open* fails; so will **TIOCSTI**

kill(2)

```
int kill(int pid, int signo)
```

Sends signal number signo to process pid

- sender's RUID or EUID must match receiver's RUID or saved UID (except if superuser)
- pid = 0 sends to all processes of same process group
- pid < -1 sends to all processes with process group id of | pid |
- pid = -1 sends to all processes with RUID equal to sender's EUID; if EUID = 0, goes to all except *init*

link(2)

```
int link(char *name, char *newname)
```

Creates another directory entry for name called newname

- » Both names must be on the same file system
- » Superuser can do link to directory
- » *newname* cannot exist

Means that file system really a general graph, not a tree

read(2), *write*(2)

```
int read(int fd, char *buf, int nchars)
int write(int fd, char *buf, int nchars)
```

- File access permissions not rechecked
- Tied to file descriptor, not name
- Can do this to deleted file
  - ... since the file object is not deleted until both the file name is deleted **and** all file descriptors for that file object are closed

Secure Temporary File

create file, open for reading and writing (descriptor *fd*)

delete file (use *unlink*)

as file is open, its directory entry is removed but the file is not yet actually deleted (only files not open used can be deleted)

write data to the file

rewind the file

do this with *fseek* or *rewind*; ***do not*** close andre open it, or it will go away!

read data back from the file

close the file

this will delete it automatically

Advantages and Disadvantages

- file cannot be accessed by any other user
  if they can get to the raw device and find the inode, they can get the data directly; but that means you're compromised anyway
- at end of program, temp file automatically deleted
  » good: ciel cleanup automatic
  » bad: may make PM analysis harder on abnormal termination

\+ race condition eliminated

– hides use of disk space
  » you see it is gone, but not where

rename(2)

Problem: how to atomically move a file

Why? Replacing password file

System crash could leave no password file

```
int rename(char *oldname, char *newname)
```

Removes *newname*, names *oldname newname*

Newname is guaranteed to exist even if system crashes

signal(2 or 3)

```
void (*signal)(int signo, int (*func)(int signo))
```

On some versions of the UNIX system:

setuid program dumps core ⇒ core file owned by owner of setuid program

Catch all signals here to prevent such a dump

Note: not possible on Version 7 as on interrupt, trap reset to default value, *then* handler called

On these systems, you can ignore signals, though

More on Signals

Why prevent core dumps?

- If world writable, attacker may be able to trick programs into executing commands as you
- If not, may contain sensitive data (like your password or secret cryptoigraphic key)

More on Signal

Signals like

- **SIGTSTP** stop signal from keyboard
- **SIGTTIN** stop on background read
- **SIGTTOU** stop on background write

suspend program

Do not rely on data files across these if they,or any ancestor directory, can be modified by untrusted users.

stat(2)

```
int stat(char *path, struct stat *buf)
```

Returns information (mode, last mod time, etc.) about file

If path is symbolic link, returns info about what link points to

Use *lstat* for info about the link itself

Use *fstat* to do this with a file descriptor

Example

```
if (lstat("/usr/spool/lpr/queuedfile", &stbuf) < 0)
    … error handling …
if ((stbuf.st_mode&S_IFMT) == S_IFLNK)
    … it's a symbolic link …
… it's not a symbolic link …
```


stat(2) Races

Warning: *fstat, stat* and *lstat* may present race conditions if:

- the file (or any of its ancestor directories) is writeable by an untrusted user
- taking some action is based on the file characteristics returned by these calls; and
- any reference is by name, not file descriptor
  - This means the other system call involved too

Safe: use file descriptors for all system calls involved

umask(2)

```
int umask(int newumask)
```

Resets process umask

Note: *newumask* is interpreted by rules of C, so don't forget leading "0" for octal numbers!

utimes(2)

```
int utimes(char *file, struct timeval tvp[2])
```

Changes time of last access (r/w) and update (w) of file

Only owner, superuser can issue this call

... but anyone who can write to disk, memory can change times in inode

Does not change inode modification (creation) time

... but anyone who can write to disk, memory can change this time

crypt(3)

```
char *crypt(char *key, char *salt)
```

Useful for password validation

- *key* is cleartext password
- *salt* is first 2 chars of hashed password
  - can just give pointer to hashed password, as only first 2 characters used
- hash of key with salt is returned

Password Testing

This returns 1 if *given* is correct password, else 0

```
int ispassword(char *given, char *hash)
{
  return(strcmp(hash, crypt(given, hash) == 0)
}
```

Memory Use

Note: cleartext password left in memory

Bad news if there's a core dump, so …

```
  for(g = given; *g; g++)
      *g = '\0';
```

Can also use *bzero*(3) or *memset*(3) if you know that the password is under some specific length:

```
(void) bzero(given, sizeof(given))
```


getusershell(3)

```
char *getusershell(void)
```

If your program needs a shell, use environment variable **SHELL** but first check it is legal

Otherwise you might exec something you don't plan to

```
while((sp = getusershell()) != NULL)
   if (strcmp(proposedshell, sp) == 0)
      …it's okay …
… it's not a legal shell …
```

mktemp(3), *mkstemp*(3)

`char *mktemp(char *template)`

This makes a unique file name

- Race condition between making file name and opening it in program

`int mkstemp(char *template)`

Like *mktemp*, but returns file descriptor of opened temp file

- Avoids race condition in program; may or may not eliminate race condition completely (depends on implementation)

Pseudo-Random Number Generation

`int rand()`

Generates a pseudorandom integer between 0 and 2147483647 (= 2^{31}—1)

Warning: low order bits not very random

Use *rand48*, *random* instead. Even these are not suitable for cryptographic purposes, though

Seeding the PRNG

Do *not* use time of day, process ID, or any function of known (or easily obtained) information

Attacker can guess the seed, and regenerate the sequence, and use that as a key to regerate the relevant random numbers.

Programming Tip: Good Style

- use a system like *lint* to check your code

  If using ANSI C, the GNU compiler has many wonderful options that have a similar effect; I recommend -Wall -Wshadow -Wpointer-arith -Wcast-qual -W
- test using random input and any bogosities you can think about

  See the marvelous article "An Empirical Study of the Reliability of Unix Utilities," by Miller, Fredriksen, and So in *Communications of the ACM* **33**(12) pp. 32-45 (Dec. 1990)

Example programs/functions

- *lsu*, program to give user privileges of a restricted account
- *mpopen*, function to run *popen* or *system* safely
- *settcpdump*, program to give *tcpdump* setuid setting

lsu Suite

lsu, *su*, *nsu*

A suite of programs to implement a new version of *su* and a group account manager *lsu*

- *lsu*

  Allow a user to *su* to a second account with knowledge only of his/her password

- *nsu*

  Like *su*, but HOME and USER environment variables are **always** reset

Design Considerations #1

Principle:

- separation of privilege

  Constrain access upon: authorization, time place

Implementation:

- use an access control file (see “lsu/perm.c”):

userid *userlist* *locations* *times*

when they can do it
which ttys they can(not) use
who can change to that account
account to change to

Design Considerations #2

Principle:

- least privilege

  Cannot require this but instead strongly recommend … *do not use this to control access to the superuser account*

Why:

- superuser can alter access control file, but no-one else can (the program enforces this; see function chkperm() in file “lsu/perm.c”, lines 209-301)

Design Considerations #3

Guideline:

- changing privileges should be an auditable event

  this means it should be logged

Why:

- in case there is a need to determine who accessed a particular account using any of this suite's programs, the log can tell who accessed what when.

Implementation:

- see the file “lsu/log.c”

Design Considerations #4

Guideline:

- changing should be traceable to an individual

  Not possible to enforce, but it can be enforced to the granularity of a single account.

Implementation:

- only users of specifically authorized accounts may change to a specific account (see the routine perms() in lsu/perm.c, lines 23-176); note a wildcard mechanism is available (see isinlist() in lsu/util.c, lines 64-113)

Design Considerations #5

Principle:

- separation of privilege again

  how can we be sure the user of *lsu* is authorized to use the account under which *lsu* is being run?

Implementation:

- require the user to supply the correct password for the account being used (*lsu*) or the new account (*su*, *nsu*) (see line 118 in “lsu/lsu.c”, which call chkpasswd() (“lsu/perms.c”, lines 197-203), which call vfypwd (“lsu/util.c”, lines 115-142)

Design Considerations #6

Guideline:

- protect against strange environments

  The **PATH** and **SHELL** must be properly set, especially if *su*ing to *root*

Implementation:

- simply reset both to a pristine state; which depends on the specific type of system being run (see “lsu/sysdep.h”, the macro **LSUPATH**), and the routines getshell(), envdoit(), and chkpath() in “lsu/lsu.c”, lines 230-381)

Notes

- User identity obtained from *getpwuid*(*getuid*()), not *getlogin* (see lines 213-223 of “lsu/lsu.c”)
- No indication of why access is denied if it is; that way, you can't use these programs to guess passwords
- Note you can log even after the setuid to new identity (which may not be *root*) because the log file is still open, and access is checked *only* at open (but line 172 of “lsu/lsu.c” may fail)

More Notes

- Note the use of *execve* (“lsu/lsu.c”, line 166) to reset the new environment
- Were I to do it again, I would change the environment check to clear everything, and reset the **umask**, **IFS**, **SHELL,** and **PATH** (and any **LD_** variables or their ilk) to *known* values that included only *trusted* directories. Not done at the time because we needed to preserve the user’s existing environment as much as possible (and all these users were trusted)

Some Reflections

Is this the best way to solve the problem?
First, what do we want?
- How would we do it on a really secure system?

Then, how can we do it?
- Should we use setuid/setgid or something else?

Reference Monitor

A security mechanism sitting between the program and the resource being protected:

- tamperproof
- complete (*ie*, always invoked)
- verifiable

Applications to UNIX System Programming

Last implies:

- the privileged code should be as *small* and as *simple* as possible
- code accessing the resource should be in a separate module

Privileged Servers

Create a privileged server to access and manipulate the resource

+ isolates all privileged code from the application or system program
+ need no longer worry about *changing* privilege

  That is, user environment is no longer relevant as all manipluations are done under the server's environment

+ other systems can use it, too

More Privileged Servers

– Lots harder to assure that data sent over a network is authentic and unmodified than to give such assurances for data internal to the computer

In other words, there is a direct path from the prigvileged system program to the kernel, so in an attack either the kernel or the program must be compromised; with a server, the attacker can now compromise the server and, if it is on a network, this is quite easy …

– Another server to feed and care for (increasing administrative load)

– other systems can use it, too

Compartmentalization

Whenever a setuid program is necessary:

- isolate all code that needs to be privileged into a small module
- write a small program to implement these functions

  You also have to put any special access control in here, too
- make your program not setuid and the small one setuid, and have your program invoke this small setuid program

What UNIX Systems *Really* Need

A way to make some modules (functions, whatever) within a program privileged without making the entire program privileged

Applying This to *lsu*

- Why not a server?
  - Idea: have the server execute the commands for us
  - Problem: network authentication problem too hard
- Compartmentalization
  - All checking and resetting done in *getshell*() and its minions
  - Good modularization throughout

mpopen

Goal: provide a safe version of *popen*(3)

Implementation: reset environment completely

Example:

```
setproc("PATH=/bin:/usr/bin");
setproc("IFS=' \t\n'");
setproc("HOME");
pp = mpopen("date", "r");
```


Design Consideration #1

Server or routine?

- Written as function because server too complex due to authentication problem

Compartmentalization

- Tight; 5 routines do everything, all are very small

mpopen, mpclose	set up call to (or wait for) child
schild	invoke child, reset environment and file descriptors
setenv, setumask	reset environment

Design Consideration #2

Guideline: Fail-Safe Defaults
Defaults provided for **PATH, SHELL,** and **IFS**
Caller can override these

See "mpopen/setproc.c", lines 9–12; overriding is done in mpopen(), lines 53–84

Design Consideration #3

Guideline: Environment reset completely

Use of *execve* in schild, along with closing of all unused file descriptors

See lines 38–44, 63 and 64 in schild()

settcpdump

Goal: need to make a specific program setuid to root

- Only 3 users (*a*, *b*, and *c*) will ever compile and run *tcpdump*
  - All are trusted users

Problem: if anyone else finds this, they can run it too …

Implementation Consideration #1

Goal: assume *a* is using it. How can we keep him from being tricked into making an arbitrary setuid to root program?

Approach: check to be sure *tcpdump* is a regular file that is executable by all and is newer than 1 minute old, and only owner and group can write to ancestor directories.

Problem: *a* can still be tricked, but window of vulnerability is very small

Implementation Consideration #1 (con't)

Use *lstat*(2)to:

- check for owner (lines 91–95)
  be sure the runner is the owner
- check for file type (lines 96–100)
  be sure the file is a regular file (not a symbolic link)
- check for age (lines 108–113)
  be sure time of last modification is under 1 minute old

All lines are in main() in "settcpdump/settcpdump.c"

Implementation Consideration #2

Who can write the directory?

- Check permissions not only of current directory but also of all ancestors
- If anyone other than the runner or his/her group can write, exit

See lines 115–125 of main(), and issafedir() in "settcpdump/issafedir.c"

Implementation Consideration #3

Goal: be sure one of *a*, *b*, *c* is running the program

Approach: use *getpwuid*(*getuid*()) to get runner; after verifying it is allowed used, validate with password. Note on error, password is required anyway

See lines 55–81 in main() in "settcpdump/settcpdump.c"

For More Information

Kochan and Wood, *UNIX™ System Security*, Hayden Books ©1985; ISBN 0-8104-6267-2

Rather dated, and quite specific for System V; but it's the only book with anything substantial for secure programming

Ferbrache and Shearer, *UNIX Installation, Security & Integrity*, Prentice-Hall ©1993; ISBN 0-13-015389-3

Good overview of functions, but limitedto those; no design principles