Writing Safe Setuid Programs

Matt Bishop

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

phone (530) 752-8060
email bishop@cs.ucdavis.edu

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 1

Why is this hard? A few 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

Matt Bishop
Dept. of Computer Science
University of California at Davis

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 3

These are from Saltzer and Schroeder:

least privilege: need-to-know

fail-safe defaults by default, deny

economy of mechanism: KISS principle

complete mediation: check every access to an object

open design: don't depend on secrecy of design giving additional security

separation of privilege: make access dependent on multiple conditions, not just one

least common mechanism: minimize sharing

psychological acceptability: security mechanisms shoud be as easy to use as not to use; difficult ideal to approach, so come as close as possible

Users and Groups

Real UID, GID:	UID, GID of user running program
Effective UID, GID:	UID, GID of user with whose privileges the process runs
Login/Audit UID:	UID of user who originally logged in
Saved UID, GID:	UID, GID before last change by program
Primary GID:	GID assigned at login
Secondary GIDs:	GIDs of groups to which the UID belongs

Matt Bishop
Dept. of Computer Science
University of California at Davis

Warnings:

getaudit(), getlogin() common for audit/login ID, but be sure getlogin is the right one!

Some systems do not allow direct access to the saved UID or GID

Setting the UID sets the effective UID unless it’s root; both real and effective UID are set. You can use separate system calls to change either.

When getting information about user or group, the getpwuid, etc. functions return the first matching entry in the passwd or group databases. This may or may not be what you want.

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 5

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

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, as you assume *root* is trusted and users aren’t
- in practise, not very, as you need to guard against poorly set up *root* 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*!

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 6

What could be done?

- 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

Riules of thumb:

If no need to log in, use group (not user), as groups generally more restricted than owner

If group compromised, usually much less dangerous; this is due to usual system configuration; not inherent in the system

Application of privilege of least principle

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 ...

Matt Bishop
Dept. of Computer Science
University of California at Davis

But setgid does not guarantee one can't do nasty things; it's usually a matter of degree …

This attack is hard and takes some knowledge of the output of *ps* to interpret. Tricking *ps* into reading the data is easy; interpreting the output is the hard part.

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

This applies to the user environment as well; we'll get to that later.

Overflows

- *login*, V6 UNIX (apocryphal?)
- *fingerd* as exploited by the Worm
- *syslogd*, *identd*, …
- lots of program argument lists

All cail to check bounds adequately

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 9

Actually, these are all different …

- login bug changed data in the data segment
- fingerd and the rest overwrite the stack

Works on RISC systems; just requires some more work

One vendor made the stack pages non-executable; but many programs *malloc* space for input or arguments, and data on the heap could be executed …

Handling 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 in the manual for those which handle arrays and do not check length

» checking for termination character is *not* enough

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 10

Assume any input (or file names, or environment variable values, or arguments, …) supplied by the user or under the user's control will be set to cause problems.

In general, don't trust input to be of the right length or form. Assume it could overflow *any* buffer, and program defensively!

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) ...
```

Say host name resolves to

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

Command executed is

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 11

The user may control non-obvious things:

- for network services, the user can control anything from the network

In the above example, theprogram trusts the results of *gethostbyaddr*; it shouldn't.

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 12

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

Environment 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

Matt Bishop
Dept. of Computer Science
University of California at Davis

Where is the privilege?

- vi is not setuid to *root*; you don't need that to edit your files
- expreserve is setuid to *root* as 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.

Matt Bishop
Dept. of Computer Science
University of California at Davis

Apparent lesson (it's one of the real ones …)

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

  Instead of resetting **PATH,** change

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

  to

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

- This by itself is not enough, however ...

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.

Matt Bishop
Dept. of Computer Science
University of California at Davis

You want to disable all environment variables, and enable only those you need -- after you have senity checking. Principle of fail-safe defaukkts.

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

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

WRONG

```
% 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`

Matt Bishop
Dept. of Computer Science
University of California at Davis

This is a very common error (one of my early -- 1985 -- TRs on the subjecthad it).

Note *system* spawns a Bourne shell, then executs te command.

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 ...

Matt Bishop
Dept. of Computer Science
University of California at Davis

This is somewhat system dependent …

What to do? Use *execve*(2) and 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.

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

Similar problem: when dynamic loading is used and load path is under user’s control.

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 18

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

So, build a dynamic library with your own version of *fgets.o*:

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

and put it into a library *libme.so* in current directory. Then, execute the following

```
% LD_PRELOAD=.:$LD_PRELOAD
% setuid_program_calling_fgets
#
```

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 …

Matt Bishop
Dept. of Computer Science
University of California at Davis

There's a catch: the program can't just ignore the variables, it must purge them from its environment lest they be passed to a non-setuid subprocess running on behalf of the setuid process. Example: /bin/login spawning /bin/sync. This was a Sun bug for a time.

Because of all this, I recommend that security-related code be statically linked. Dynamically linked code can be secure, but it is affected more by the environment and the run-time libraries than is static code.

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 20

This is critical, as security is in large part knowing (and validating) your assumptions.

Moral of all this?

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

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!

Matt Bishop
Dept. of Computer Science
University of California at Davis

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

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 and is much smaller, but still exists

Matt Bishop
Dept. of Computer Science
University of California at Davis

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

The Smaller Hole

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*

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 23

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

Race Condition Problem

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

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

Note: order of fopen and access can be switched and same problem occurs.

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!!!

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 25

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

ReUse *trustfile* from

ftp://nob.cs.ucdavis.edu/pub/sec-tools/trustfile.tar

File Descriptors and Subprocesses

```
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*

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 26

These are not closed across fork or exec

- Threat is when privileged parent opens sensitive file and then spawns a subshell

File Descriptors and Privileges

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

File descriptors are essentially capabilities; once you have one, you can read/write the file eve if it is deleted.

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 28

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

In general, don't use setuid scripts; 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

Keep It Small and Simple (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*

Matt Bishop
Dept. of Computer Science
University of California at Davis

Good example is shell scripts. Even if the kernel bug above is fixed, shells often base actions uipon the name of the shell; if the first char of arg 0 is "-", it's a login shell.

Just write a 4-line C program to do this, and call the subsequent shell "-*xyz*".

Other interpreters (*awk*, etc.) have this same problem.

Possible Side Effect of Shell Scripts

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

Bb Morris thinks this is either apocryphal orcomes from a local modification of *su*(1), as he wrote the V6 *su* and did not put this in.

Error Recovery

With privileged programs, it's very simple:

<u>DON'T</u>

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

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**

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* .

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 33

Checking the cause of an error:

```
#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)

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

Matt Bishop
Dept. of Computer Science
University of California at Davis

Now for some odds and ends …

- 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

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))
```

Matt Bishop
Dept. of Computer Science
University of California at Davis

Also, clean out files by overwriting if they contain sensitive data; on some systems, *trunc*(2) or *ftrunc*(2)zaps the data, too.

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.

Matt Bishop
Dept. of Computer Science
University of California at Davis

Also, check quality of PRNG if it’s used for anything sensitive, like cryptographic keys.

Bug in a routine on some systems:

```
int rand()
```

Generates a pseudorandom integer between 0 and 2147483647 (= 2 —1)

Warning: low order bits not very random

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

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)

Matt Bishop
Dept. of Computer Science
University of California at Davis

Slide # 37

Conclude: we need to face this problem. As the good doctor (Seuss) says,

But I've bought a big bat.
I'm all ready, you see;
Now my troubles are going
To have troubles with *me*!