Writing Safe Setuid Programs

Matt Bishop

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

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

Theme

Using standard robust programming techniques can greatly improve the quality of security-related code, which involves:

- 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

Basics of Robust Programming

- Be paranoid
  - » Principles of least privilege, least common mechanism
- Assume maximum stupidity
  - » Principles of fail-safe defaults, separation of privilege, psychological acceptability
- Don't hand out dangerous implements
  - » Principles of least privilege, fail-safe defaults, complete mediation, economy of mechanism
- Worry about cases that "can't happen"
  - » Principles of least privilege, open design, separation of mechanism

Six Implementation Problems

- Unstated or implicit assumptions
- Unknown interactions with system components
- Numeric or buffer (array) overflow
- Altering and/or deleting files
- Race conditions
- Invoking a subprocess

Unstated or Implicit Assumptions

Goal: read any location in kernel memory

ps accesses process table by:

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

ps is typically setgid to group kmem so it can read the memory device files

User can specify where *vmunix* file is

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

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 writable 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

Arguments and Return Values

Check that arguments are reasonable

Example: failure to check that pointer is in user space in a kernel division allowed users to overwrite their UID with 0

Check return values

Example:

```
int validate(char * user);
/* ... */
validate(user);
```

Errors

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 …

Use `errno` to disambiguate cause of failure

Morals

- Explicitly state assumptions
- Validate arguments
- Never assume a function or system call succeeds
- Don't make assumptions to handle errors; if you cannot determine the cause of failure, or do not know how to recover safely, ***stop***

Unknown Interaction with System Components

Get IP address 55.5.12.1.2; want host name

Use *gethostbyaddr*, 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
```

User Specifying Input

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

Example: 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

Shell Scripts

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

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

The Obvious Fix

Problem: Dynamic loading allows an unprivileged user to alter 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 …

Morals

Extension of first item …

- Minimize interactions; make the program as self-contained as possible
- Validate all results from others (processes, users, etc.) unless you trust the source
- Be sure your trust is well placed

Numeric or Buffer (Array) Overflows

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

All fail to check bounds adequately

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

Numeric Overflow

Year 2038 problem …

`2147508847` is `Tue Jan 19 03:14:07 2038`

`2147508848` is `Fri Dec 13 20:45:52 1901`

So overflow can foul up the time

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

Moral

- Check all array manipulations for potential overflows
- Check all pointer manipulations for potential overflows
- Check all numeric operations for potential overflows *and underflows*

Altering and/or Deleting Files

Watch out when you *open* a file for writing:
open(filename, O_WRONLY|O_CREAT, 0644)
creates a file, but will clobber an existing one
open(filename, O_WRONLY|OCREAT|O_EXCL, 0644)
won't clobber an existing file.

Symbolic links? Check your system!

Morals

- Watch out when you create a file; you may zap one that is there
- Know how programs that take pathnames handle symbolic links; does the operation apply to the *link* or to the *referent*?

Race Conditions

To check ability to access a test config file …

```
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*

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!!!

Example: 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 and reopen!

read data back from the file

close the file

this will delete it automatically

A Kernel Race Condition

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

Morals

- Ensure that you use only those objects you've checked or that you trust
- Refer to files using descriptors (not path names) whenever possible
- Be careful with temporary files
- ***Never*** make an interpreted setuid (setgid) command script

Invoking a Subprocess

At Purdue, when I was a grad student …

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*!

Environment Variables

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

Attack #1

```
$ 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.

Attack #2

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.

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 '`

Multiple Definitions

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 ...

More Environment

- *umask*
- UIDs and GIDs

  real, effective, saved, login/audit UIDs; real, effective, primary, secondary GIDs
- notion of /
- options

Morals

- No program executes independently; subprograms always carry their environment with them.
- Setuid program gives privileges for the life of the process, plus any descendants, so the owner must dictate the protection domain
- Turn off *all* environment variables; then define only those you need

Miscellaneous

- Inheriting file descriptors
- Memory and core dumps
- Pseudorandom number generation
- Style and testing

Style and Testing

- 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 of

  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)

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))
```

Seeding a 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 regenerate the relevant random numbers.

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*

The Doctor's Prescription

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