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
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
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 should be as easy to use as not to use; difficult ideal to approach, so come as close as possible
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

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
Writing Safe Privileged Programs

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!

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

Rules 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 \textit{ps(1)} Attack

\textit{Goal:} read any location in kernel memory

\textit{ps} accesses process table by:
- opening symbol table in /vmunix
- looking up location of variable \_proc

\textit{ps} setgid to group kmem
User can specify where \textit{vmunix} file is
So supply your own \textit{vmunix} and read any file that group kmem can read ...

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 \textit{ps} to interpret. Tricking \textit{ps} into reading the data is easy; interpreting the output is the hard part.
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

This applies to the user environment as well; we’ll get to that later.
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 …
Writing Safe Privileged Programs

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

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!
The user may control non-obvious things:

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

In the above example, the program trusts the results of `gethostbyaddr`; it shouldn’t.
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
Writing Safe Privileged Programs

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
```

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

Look for any code using environment variables:

```c
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:

```bash
system("IFS=\n\t ;PATH=/bin:/usr/bin;\n    export IFS PATH;command");
This sets IFS, PATH: you may want to fix more
```

**WRONG**

```bash
% 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`

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

Note `system` spawns a Bourne shell, then executes the command.
This is somewhat system dependent …

What to do? Use `execve(2)` and reset what parts of the environment you want:

```c
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.
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`:

```c
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
```

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
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.
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
- GIDs
- umask
- open file descriptors

- root directory of process
- file system paths of referenced files
- network information
- process name

Essentially, environment is the protection state of the system plus anything that affects that state
Writing Safe Privileged Programs

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!

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
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

---

One Partial Fix

use `access(2)` system call:

```c
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
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
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!!!*

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
These are not closed across fork or exec

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

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

Running this and typing

```bash
% cat <&9
```

prints the contents of `priv_file`
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 protected log file as root
» drop privileges to user
» can still log data in protected file

File descriptors are essentially capabilities; once you have one, you can read/write the file even if it is deleted.
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
Good example is shell scripts. Even if the kernel bug above is fixed, shells often base actions upon 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.
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 \textit{su} Bug (Apocryphal?)

If \textit{su} could not open password file, assumed catastrophic problem and gave you root to let you fix system

Attack: open 19 files, then \texttt{exec su root}
  At most 19 open files per process, so …

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

Bb Morris thinks this is either apocryphal or comes from a local modification of \textit{su}(1), as he wrote the V6 \textit{su} and did not put this in.
Writing Safe Privileged Programs

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

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**
Checking the cause of an error:

```c
#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 \textit{fd})
delete file (use \textit{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 \texttt{seek} or \texttt{rewind}; \textbf{do not} close and re open it, or it
will go away!
read data back from the file
close the file
    this will delete it automatically

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 …

\[
\text{for}(g = \text{given}; \ast g; g++) \\
\ast g = '\0';
\]

Can also use \texttt{bzero}(3) or \texttt{memset}(3) if you know
that the password is under some specific length:
\[
(\text{void}) \text{bzero} (\text{given, sizeof(given)})
\]

Also, clean out files by overwriting if they contain sensitive data; on
some systems, \texttt{trunc}(2) or \texttt{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 regenerate the relevant random numbers.

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

Bug in a routine on some systems:

```c
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.
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!