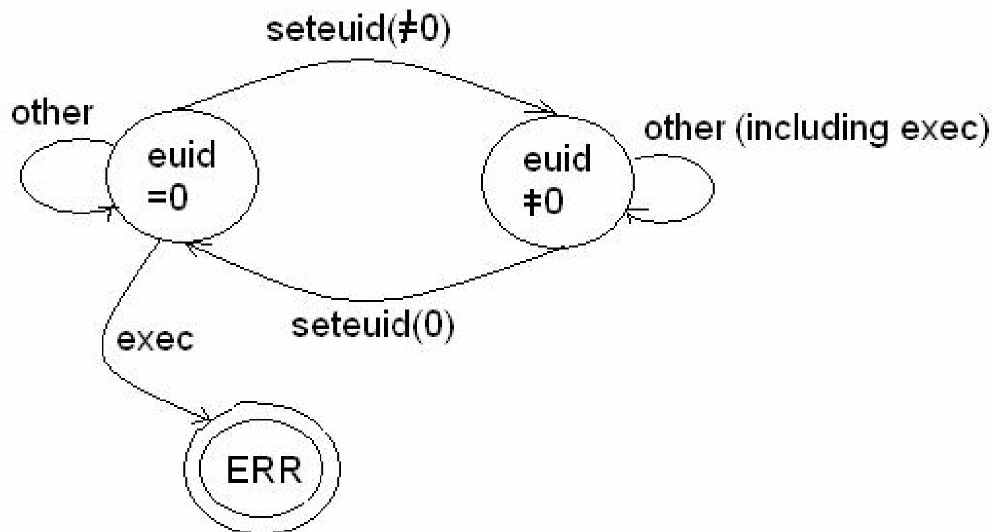
CSE 509 : System Security, Spring 2006.
Lecture Notes for 03/14/2006.

setuid/exec bugs:

Unix (and its flavours) provide a system call seteuid() to change a user's current effective user id.
- seteuid(0) : effective user id to become 0. (So, become root temporarily)
- seteuid(getruid()) : go back / drop root privilege temporarily.



Why would you drop root privilege ?
- If a trusted program wants to run an untrusted helper program.
e.g. emacs launched from cron.

A problem, could be if a user forgot to drop the root privilege.
Consider the following piece of code :

```
void runhelper()
{
        if((cmd= malloc(...))!=NULL)
                seteuid(getruid());
        exec(...);
        seteuid(0);
}
```

Now if Malloc fails, then exec is done as root.

(For getting dirty with setuid, see the paper : Setuid Demystified:
http://www.cs.berkeley.edu/~daw/papers/setuid-usenix02.pdf)

chdir/chroot:

Consider the following piece of code :

```
void switch_to_new_root(...)
{
	chroot("/var/anonftp");
	printf("Switched to new root !!");
}
```

Once the chroot is done, processes can only access files under /var/anonftp
The problem is that chroot doesn't change the CWD (current working directory) of the FTP server.

Inside OS, there is

```
  root          CWD
   /        /usr/share
```

Now, open("pics/stuff"); starts from CWD.

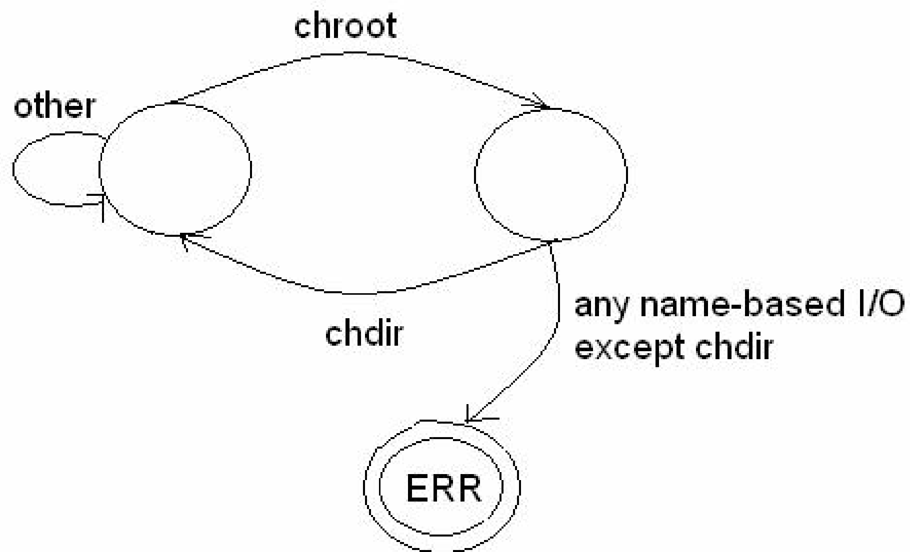
If we have issued chroot("/var/anonftp"),
it looks like this :

```
     root          CWD
  /var/anonftp    /usr/share
```

Now, another open("pics/stuff"); starts from CWD and still works !!

So, need to add a chdir("/"); after doing the chroot().
i.e. the code should look like :

```
void switch_to_new_root(...)
{
	chroot("/var/anonftp");
	chdir("/");
	printf("Switched to new root !!");
}
```



Tractor-Beaming Attack (Funky !)

Consider the following piece of code from a ftpserver:

```
jmpbuf jb;
try_transfer(...)
{
        setjmp(&jb);
        do_transfer(...);
label1:
        …………..
        …………..
}

do_transfer(...)
{
label2:
        seteuid(0);
        // do some I/O with TCP
        seteuid(getruid());
}

sig_urg_handler()
{
        longjmp(&jb);
}
```

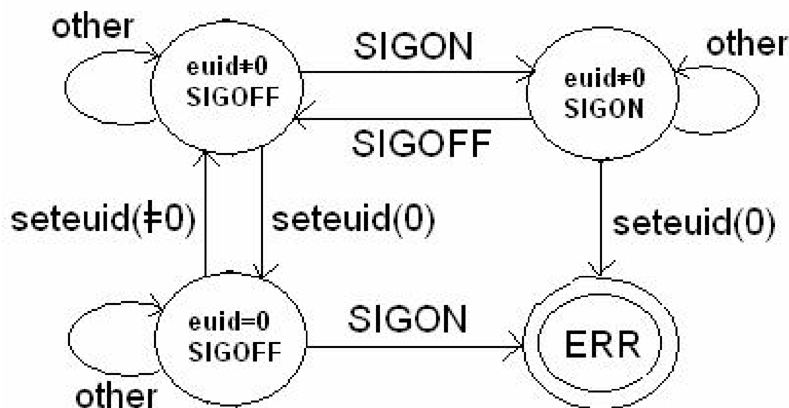
In the above code, we use setjmp - longjmp. TCP can send an urgent out of band message, and the handler for the urgent signal (sig_urg_handler()) does a long jump to

label1 in try_transfer(). What this means is that, the ftpserver is now root, since we did the seteuid(0) but did not drop the root privileges before doing the long jump.

There are several ways to fix this problem, one of which is to ignore the urgent message signal from TCP as shown below:

```
do_transfer(...)
{
label2:
        signal(SIG_URG,IGNORE);
        seteuid(0);
        // do some I/O with TCP
        seteuid(getruid());
}
```

But, our goal is not to enable programmers to do the right thing, but to disable them from doing wrong.
Lets propose a rule, when euid=0, all signals must be ignored.



For 32 signals, 2^32 states in a state machine, +1 for euid, so analyzing all signals simultaneously requires 2^33 states, but we can analyze each signal independently efficiently.

File Descriptor Attack:

For a setuid program, user controls initial state of file descriptors. You always have 3 file descriptors :
STDIN: 0
STDOUT: 1
STDERR: 2

printf("HELLO");   // hard-coded write using fd 1 (stdout).

Now, if I ran the following code with only STDIN open, then fd = 1.

```
fd=open("some-critical-file");
printf("opened critical file\n");
```

So, setuid programs should do close(0); close(1); close(2); and then do open("/dev/tty"); open("/dev/tty"); open("/dev/tty");

As we see, all the attacks above had some state info globally.

Unchecked input:

Consider the following piece of code:

```
int len;
read(networkfd, &len, sizeof(len));
buf=malloc(len);
```

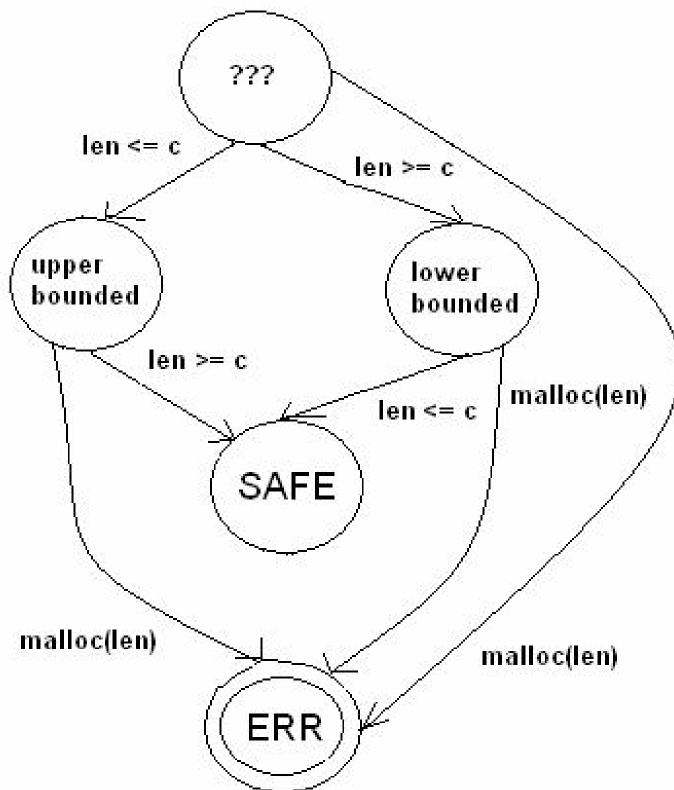
There should be a check how much read returns. (Since, read could read an amount less than that requested.)
something like:

```
if(read(...)==sizeof(len))
```

The value of len here is untrusted (comes from an external source.)
There should be a bound check on len.
something like:

```
if(read(...)==sizeof(len))
        if(len<100 && len>=0)
                buf=malloc(len);
```

So, len has state as shown by the state machine below.



This is not a very conservative approach. The state machine doesn't verify if it is bounded between the values that we need it to be bounded between.

Access / Open :
----
setuid-root program wants to open "file" if and only if invoker could open "file".

if we had something like : open("file") and the file came from an untrusted source, then the victim could possibly open anything (any malicious file).
So, there is an access/open mechanism.

```
if(access("file"))
        open("file");
```

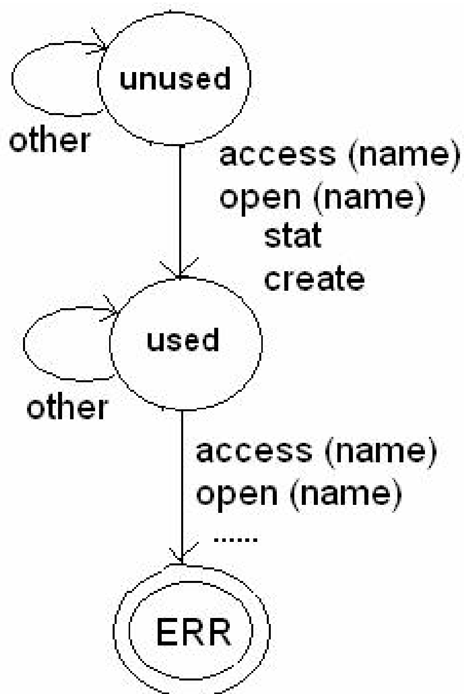
An attack on this, is if file is as follows:
"file" is owned by attacker, invokes setuid victim program. Then victim runs.
Executes access("file"), gets an "ok' from the OS, it is then scheduled out by the OS.
The attacker's code now gets control and executes rm("file") and creates a softlink to the /etc/shadow file. then it is scheduled out by the OS.
The victim code then executes the open("file") which actually ends up opening the /etc/shadow file.

Just showing the same thing, nicely :

Victim runs	Attacker runs
	file is owned by the attacker invokes setuid victim program
access("file") gets "OK" from the OS	
	rm "file" ln -s /etc/shadow file
open("file")	

So, there is no way to atomically do the access and open.
The real issue is that there are two uses on the name "file" in two different system calls between which the OS can schedule other processes/tasks.

To prevent this, we could wrap around them into a transaction.
But this enables programmers to do the right, doesn't disable them from doing the wrong.



Here are 2 interesting papers:

"Fixing Races for Fun and Profit: How to abuse atime" -
http://www.cs.berkeley.edu/~daw/papers/races-usenix05.pdf

"Fixing Races for Fun and Profit: How to use access(2)" -
http://www.csl.sri.com/users/ddean/papers/usenix04.pdf