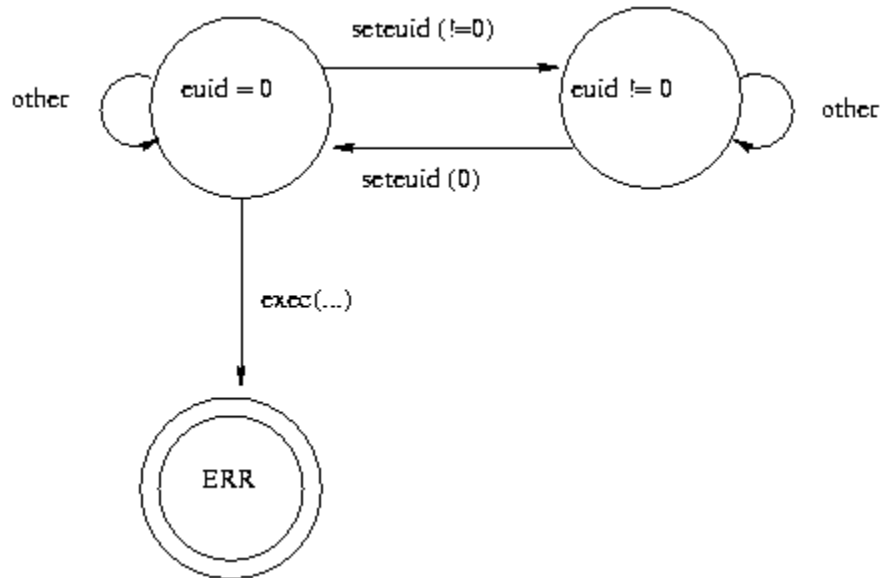


Class notes for CSE509

March 14th

setuid/exec

- setuid(0) : become root
- setuid(getuid()) : drop root privilege temporarily



Why would root drop its privilege?

Trusted program wants to run an untrusted helper program.

```
void runhelper()
{
    if ((cmd = malloc(...)) != NULL)
        setuid(getuid());
    exec(...); //if malloc failed, exec(...) might be executed with root privilege.
    setuid(0);
}
```

Chroot/chdir

```

void switch_to_new_root()
{
    chroot("/var/anonftp");
    chdir("/"); //if no this line, there might be some security problem.
    printf("switched to new root");
}

```

Explanation:

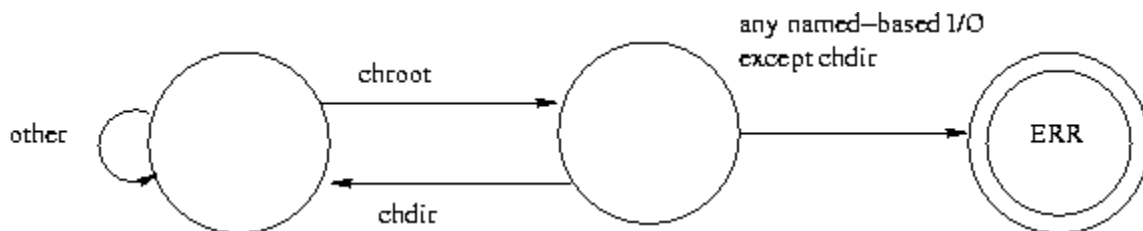
Operating system maintains root directory and current working directory, for example:

root	CWD
/	/usr/share

When we resolve a name “/pic/stuff”, OS will first lookup the root directory, then lookup “var”, then “anonftp”. When we resolve a name “pic/stuff”, OS will start resolving from the current working directory.

After the `chroot("/var/anonftp")`, process can only access file under /var/anonftp. This means that after the `chroot()` call, an `open("/", O_RDONLY)` would open the **same** directory as an `open("/var/anonftp", O_RDONLY)` call before the `chroot()`.

Why `chdir("/")` is needed? `chdir("/")` changes the CWD to “/var/anonftp”. This is to ensure that the working directory of the process is within the `chroot()`ed area before the `chroot()` call takes place. This is due to most implementations of `chroot()` not changing the working directory of the process to within the directory the process is now `chroot()`ed in. Without the `chdir("/")`, an `open("../..", O_RDONLY)` actually can open the original “/” root directory instead of “/var/anonftp”.



Tractor beaming

```

jumbf jb;
try_transfer(...)
{
    signal(SIG_URG, IGNORE);// there might be security problem without this line.
    setjmp(&jb);
    do_transfer(...);
}

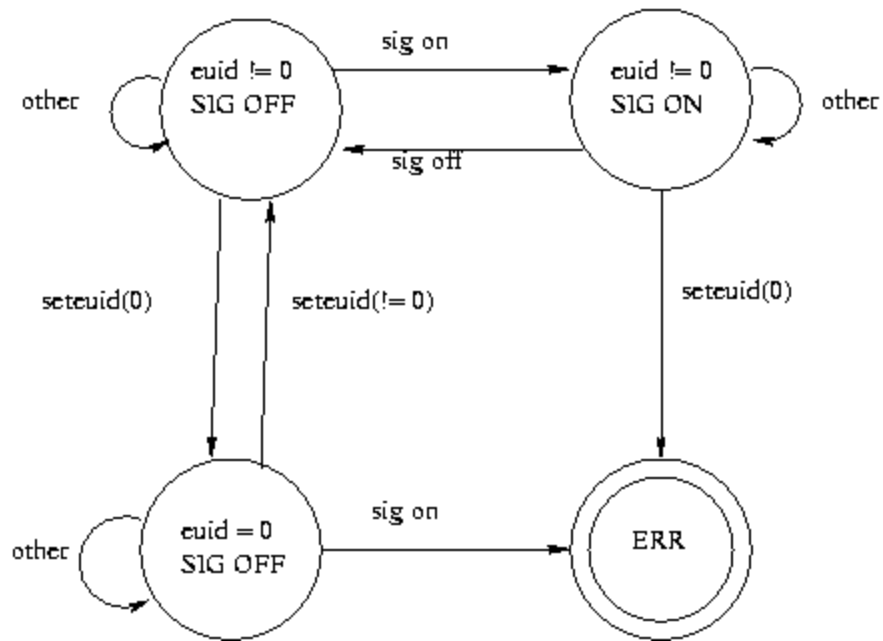
do_transfer()
{
    seteuid(0);
    //do some I/O via TCP
    seteuid(getruid());
}

sig_urg_handler()
{
    longjmp(&jb);
}

```

Without the `signal(SIG_URG, IGNORE)` line, an attacker can do the attack as following: while in the `do_transfer()` doing some I/O via TCP, give an urgent message, then the program will jump to `sig_urg_handler()` which will go to the next line of `do_transfer(...)` in the `try_transfer()` function, then the `seteuid()` in the `do_transfer()` is not executed.

Rule: when `euid = 0`, all signals must be ignored.



File Descriptor

For a setuid program, user controls initial state of file descriptors.

stdin 0

stdout 1

stderr 2

```
fd = open("some_critical_file");
printf("opened critical file \n");
//if the progrma runs with only stdin open, then fd = 1
```

```
setuid programs should
close(0); close(1); close(2);
open("/dev/tty");
open("/dev/tty");
open("/dev/tty");
```

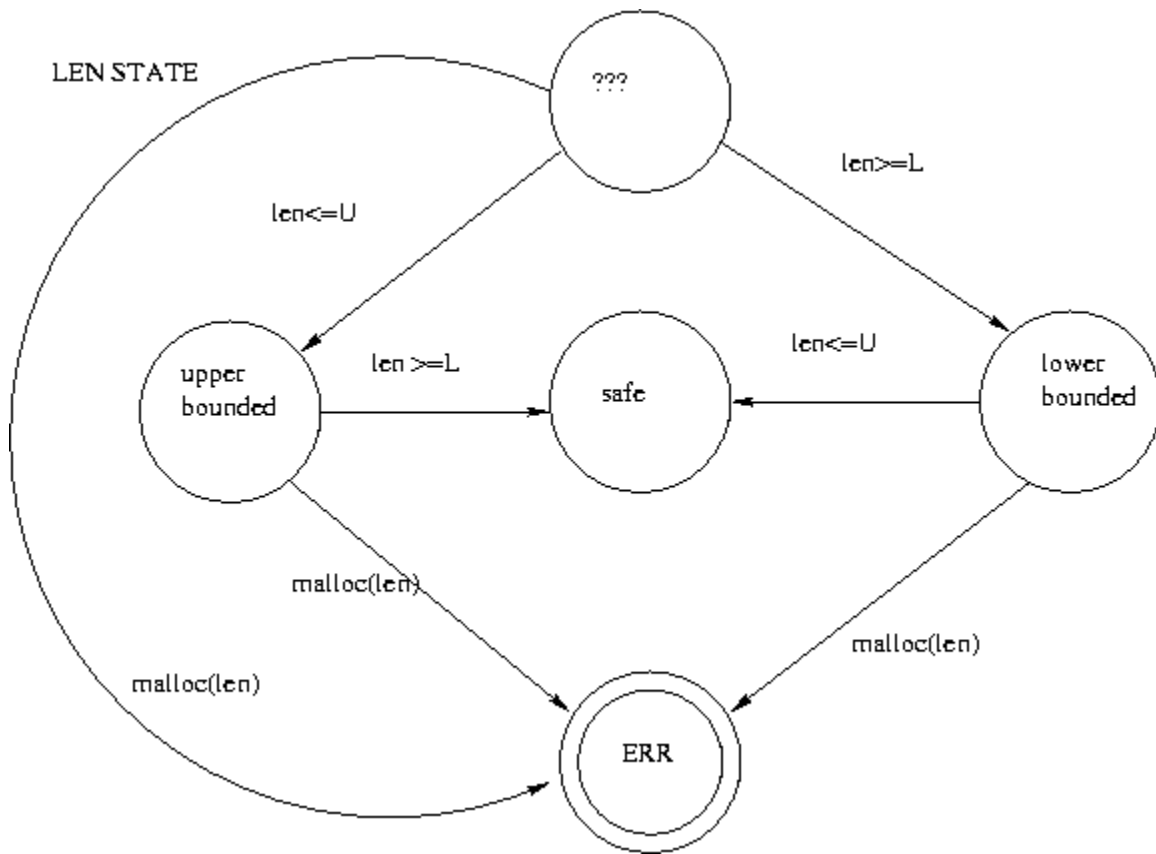
Unchecked integers

```
int len;
if (read(networkfd, &len, sizeof(len)) == sizeof(len))
    if(len<100 && len>=0)
```

```

buf = malloc(len);
//len is untrusted -> should bound check it.

```



access/open

- setuid root program wants to open "file" iff invoker could open "file"
if (access("file"))
open("file");

Attack:

"file" is owned by attacker. Attacker invoke setuid victim program. Victim runs access ("file"), which gets back "OK" from OS.

Then because of scheduling in OS, the victim might happen to be suspended at that point and scheduler schedule the attacker to run. Attacker can do the following:

```
rm "file"
```

```
In -s /etc/shadow file
```

After that, when the victim resumes running and executes the open("file"), which actually

open the shadow file.

NAME STATE

