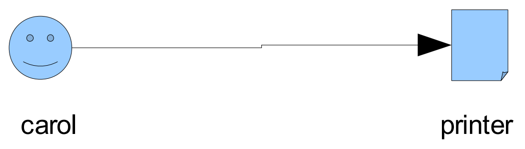
Software Based Protection

Lecture Notes 9/30
Jon Whiteaker

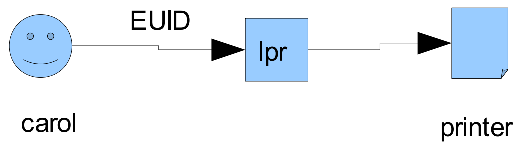
1. **Language support for privilege separation**
   - No need for multiple processes, everything can run within the same vm
   - Paper didn't show the performance breakdown for slow privilege separation with a fast language like C and fast privilege separation with a slow language like Java
     - Possible that no such comparison has been made yet

2. **Automatic privilege management**
   - Makes it to be non-bypassable since it is running in that software environment.

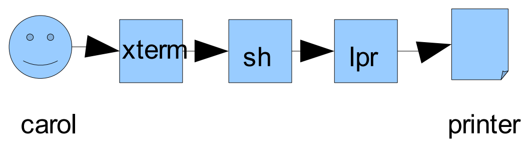
3. **Delegation problem**
   - Often times a program may actually need privileges from more than one source.
   - In the real world, if you don't have access, you e-mail someone who does. By doing this you reveal your document to the other person. If you continue to spam the other person, they may give up and just give you their login credentials, leading to a less secure system.



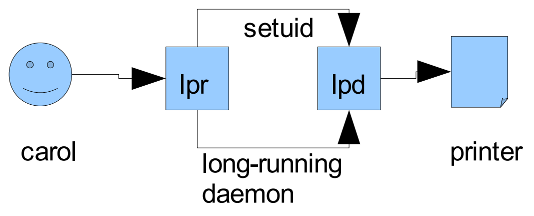
Conceptually, we usually think of of each user either having access to a printer or not.



However, usually, a user has to go through the printing program (lpr), passing along its authority.



Furthermore, there are usually multiple programs that the user goes through before it even gets to lpr, each passing along authority.



However, in OS access control lists (ACL), individual users might not have access directly to a printer (especially if the sysadmin wants to limit the number of pages, etc.), instead they have to elevate the privileges somehow. This can be done by accessing the printer through a long-running daemon, or by using setuid.

- **setuid**
  - setuid is a way to run a program with the (usually higher) permissions of the owner of the file. A flag is set in the file to allow this, and when the program is run the permissions can be temporarily raised to the file owner's permissions.
  - However, the permissions and program state/context are still passed from the executor, so the program can still run using those permissions.

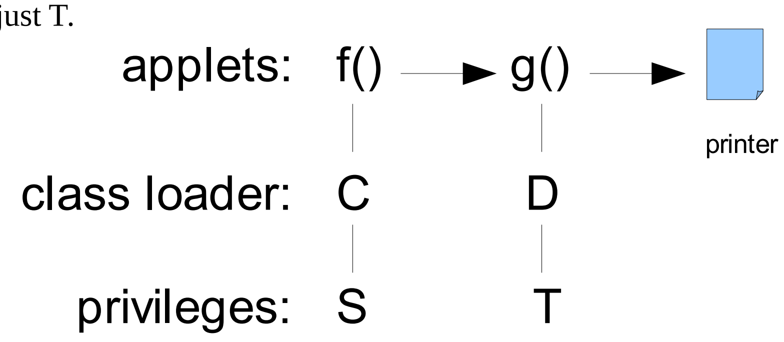
euid (access control)	uid (accounting purposes)	seteuid(carol(uid)/root(owner))
0	carol	init
carol	carol	seteuid(uid)
0	carol	seteuid(0) enable privileges
0	carol	open(printer , rw) ...
carol	carol	seteuid(uid) disable privileges

- **Long running daemon**
  - A long running daemon is usually started at the startup of the operating system, executed as root, thus running with full privileges. In order to access the the daemon, programs use inter-process communication (IPC) to make requests (such as print jobs) to the daemon.

- **Daemon vs. setuid**
  - setuid tends to have more security holes in this type of situation.
    - Since setuid inherits context from invoker, one possible attack would be to change a library path before executing the process.

    -
      ```
      int fd = open("/etc/passwd", O_RDWR);
      if (fd < 0){
           fprintf(stderr, "Couldn't open file ...");
           exit(1);
      }
      ```
      If you close stderr beforehand (file descriptor 2), then fd will now be stderr, meaning that the error case will actually overwrite the passwd file since it is outputting to stderr.

- **Java stack inspection**
  - If enable/disable privileges has not been introduced, then all callers on the stack must hold the privileges for the resource. In terms of the diagram below, that is S ∩ T.
  - With enable/disable privileges, Java will inspect the stack like before, except that this time it will additionally disallow privileges if they have been explicitly disable, and it will stop traversing the stack once it finds enabled privileges. In terms of the diagram below, that is (S ∩ T) ∪ T, or more simply, just T.



- **Java enable/disable privileges vs. setuid**
  - Offers similar concept of specifying separating areas of code that need privileges from those that don't.
  - Java offers a more fine-grained solution, available for every object instead of the simply all or nothing privileges setuid offers.
  - In Java, there is no need to explicitly disable privilege, if the applet ends then so do the privileges since they are calculated based on the current running applet, not like setuid where if you forget to disable privileges, they will propagate up the stack.

- **Problems with Java stack inspection**
  - Problems occur when the control flow is not evident
    - Call stack may not have a record of follow on code
    - Event driven programming
  - Compiler optimizations can break stack inspection through small optimizations that modify the stack for faster runtime.

- **Unix vs. setuid vs. daemon vs. Java**

f --> g max S T	**Active privileges in f**	**Active privileges in g**	**Max g privileges**	**Problems**
1) Unix program execution	S	S	S	Can't get additional privileges
2) Unix setuid exec	S	T	$S \cup T$	luring attack (elevation of privilege)
3) IPC to a daemon	S	T	T	luring attack (also can be $S \cup T$ in Windows)
4) Java stack inspection	S	$S \cap T$	T	-

  - Unix program execution doesn't allow a way to get any additional privileges, and conversely, setuid allows for too many additional privileges to be garnered.
  - Using a daemon is a more reasonable, but it is still possible to do a luring attack, leaving only stack inspection.