

Adatbiztonság

Access control

*Some slides: William Stallings and Lawrie Brown
Computer Security: Principles and Practice*

Lecture slides by Lawrie Brown

Dr. Bencsáth Boldizsár
adjunktus

BME Híradástechnikai Tanszék
bencsath@crysys.hit.bme.hu



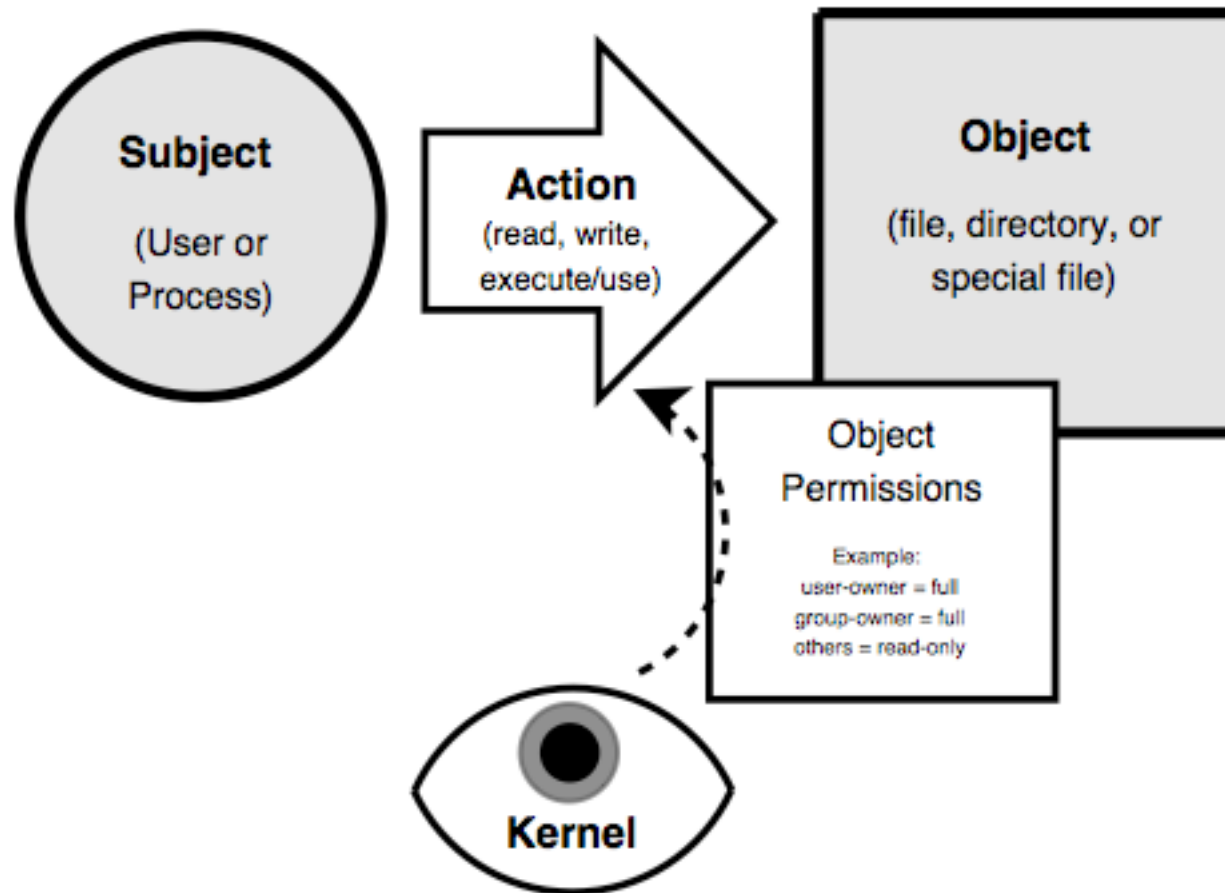
2011. április 5.
Budapest

- Linux has evolved into one of the most popular and versatile operating systems
- many features mean broad attack surface
- can create highly secure Linux systems
- will review:
 - Discretionary Access Controls
 - typical vulnerabilities and exploits in Linux
 - best practices for mitigating those threats
 - new improvements to Linux security model

Linux Security Model

- Linux's traditional security model is:
 - people or processes with "root" privileges can do anything
 - other accounts can do much less
- hence attacker's want to get root privileges
- can run robust, secure Linux systems
- crux of problem is use of **Discretionary Access Controls (DAC)**

Linux Security Transactions



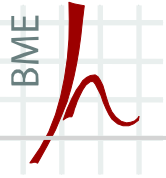
- in Linux *everything* as a file
 - e.g. memory, device-drivers, named pipes, and other system resources
 - hence why filesystem security is so important
- I/O to devices is via a “special” file
 - e.g. `/dev/cdrom`, `/dev/console`, `/dev/sda`
- have other special files like named pipes
 - a conduit between processes / programs

Users and Groups

- a user-account (user)
 - represents someone capable of using files
 - associated both with humans and processes
- a group-account (group)
 - is a list of user-accounts
 - users have a main group
 - may also belong to other groups
- users & groups are **not** files

Users and Groups

- user's details are kept in `/etc/passwd`
`maestro:x:200:100:Maestro Edward`
`Hizzersands:/home/maestro:/bin/bash`
- additional group details in `/etc/group`
`conductors:x:100:`
`pianists:x:102:maestro,volodya`
- use **useradd**, **usermod**, **userdel** to alter



File Permissions

- files have two owners: a user & a group
- each with its own set of permissions
- with a third set of permissions for other
- permissions are to read/write/execute in order user/group/other, cf.

```
-rw-rw-r-- 1 maestro user 35414 Mar 25 01:38 baton.txt
```

- set using chmod command

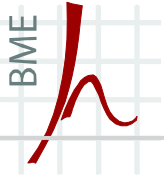
- read = list contents
- write = create or delete files in directory
- execute = use anything in or change working directory to this directory

- e.g.

```
$ chmod g+rx extreme_casseroles
```

```
$ ls -l extreme_casseroles
```

```
drwxr-x--- 8 biff drummers 288 Mar 25 01:38  
extreme_casseroles
```



Sticky Bit

- originally used to lock file in memory
- now used on directories to limit delete
 - if set must own file or dir to delete
 - other users cannot delete even if have write permission
- set using chmod command with +t flag, e.g.
`chmod +t extreme_casseroles`
- directory listing includes t or T flag (T=t, but not x)
`drwxrwx--T 8 biff drummers 288 Mar 25 01:38
extreme_casseroles`
- only apply to specific directory not child dirs

SetUID and SetGID

- setuid bit means program "runs as" owner
 - no matter who executes it
- setgid bit means run as a member of the group which owns it
 - again regardless of who executes it
- "run as" = "run with same privileges as"
- *are very dangerous* if set on file owned by root or other privileged account or group
 - only used on executable files, not shell scripts

SetGID and Directories

- setuid has no effect on directories
- setgid does and causes any file created in a directory to inherit the directory's group
- useful if users belong to other groups and routinely create files to be shared with other members of those groups
 - instead of manually changing its group

Kernel vs User Space

- Kernel space
 - refers to memory used by the Linux kernel and its loadable modules (e.g., device drivers)
- User space
 - refers to memory used by all other processes
- kernel enforces Linux DAC → it is critical to isolate kernel from user space
 - kernel space never swapped to disk
 - only root may load and unload kernel modules
 - The situations is very similar on windows, modules -> drivers

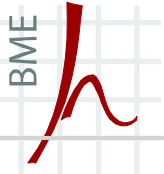
setuid root Vulnerabilities

- a **setuid root** program runs as root
 - *no matter who executes it*
- used to provide unprivileged users with access to privileged resources
- must be very carefully programmed
- if can be exploited due to a software bug
 - may allow otherwise-unprivileged users to use it to wield unauthorized root privileges
- distributions now minimise setuid-root programs
- system attackers still scan for them!

- allow attacker to cover their tracks
- if successfully installed before detection, all is very nearly lost
- originally collections of hacked commands
 - hiding attacker's files, directories, processes
- now use loadable kernel modules
 - intercepting system calls in kernel-space
 - hiding attacker from standard commands
- may be able to detect with some tools, e.g. chkrootkit
- General solution: wipe and rebuild system (including BIOS and other firmware!)

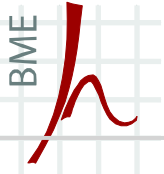
Basic Unix tools

- `cat`: Concatenate and print files
- `cd`: Change the working directory
- `ls`: List directory contents
- `chgrp <group> <file>`: Change the file group ownership
- `chmod <right> <file>`: Change the file modes/attributes/permissions
- `cp <source> <target>`: Copy files
- `mkdir`: Make directories
- `touch`: Change file access and modification times
- `id`: Return user identity



Making new directories and files: the umask

```
boldi@rivest:~/access_control $ ls -la
total 24
drwxr-xr-x  2 boldi users 4096 2010-10-26 12:58 ./
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../
boldi@rivest:~/access_control $ mkdir test
boldi@rivest:~/access_control $ ls -la
total 28
drwxr-xr-x  3 boldi users 4096 2010-10-26 12:58 ./
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../
drwxr-xr-x  2 boldi users 4096 2010-10-26 12:58 test/
boldi@rivest:~/access_control $ umask
0022
boldi@rivest:~/access_control $ touch test.txt
boldi@rivest:~/access_control $ ls -la
total 28
drwxr-xr-x  3 boldi users 4096 2010-10-26 12:58 ./
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../
drwxr-xr-x  2 boldi users 4096 2010-10-26 12:58 test/
-rw-r--r--  1 boldi users   0 2010-10-26 12:58 test.txt
```



Playing with umask

Umask determines what rights are set on the newly created files.

```
boldi@rivest:~/access_control $ umask -S
```

```
u=rwx,g=rx,o=rx
```

```
boldi@rivest:~/access_control $ umask
```

```
0022
```

```
boldi@rivest:~/access_control $ umask u=rwx,g=rwx,o=rx
```

```
boldi@rivest:~/access_control $ umask
```

```
0002
```

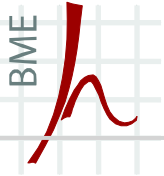
```
boldi@rivest:~/access_control $ umask u=rx,g=rx,o=rx
```

```
boldi@rivest:~/access_control $ umask
```

```
0222
```

```
(read:4 write:2 execute:1)
```

- Octal umasks are calculated via the **bitwise AND** of the unary complement of the argument (using bitwise NOT) and the permissions specified by the program: typically 666 in the case of files, and 777 in the case of directories.
- A umask set to `u=rwx,g=rwx,o=` (0007 in octal form) will result in new files having the modes `-rw-rw----`, and new directories having the modes `drwxrwx---`



An example with multiple groups

A UNIX user can be in multiple groups:

```
boldi@rivest:~$ id
uid=1000(boldi) gid=1000(boldi) groups=4(adm), 1000(boldi)
boldi@rivest:~$ newgrp adm
boldi@rivest:~$ id
uid=1000(boldi) gid=4(adm) groups=4(adm),1000(boldi)
```

Only one group is effective at a time. At starting this is the default group setting from the `/etc/passwd` file, but the user can change it with the `newgrp` command.

```
boldi@rivest:/tmp/test$ mkdir a
boldi@rivest:/tmp/test$ ls -la
total 340
drwxrwxr-x 3 boldi boldi 4096 Oct 26 13:20 .
drwxrwxrwt 3 root  root 4096 Oct 26 13:20 ..
drwxrwxr-x 2 boldi adm 4096 Oct 26 13:20 a
```

Setgid directories

```
boldi@rivest:/tmp/test$ chmod g+s .
```

```
boldi@rivest:/tmp/test$ ls -la
```

```
total 340
```

```
drwxrwsr-x 3 boldi boldi 4096 Oct 26 13:20 .
```

```
drwxrwxrwt 3 root root 4096 Oct 26 13:23 ..
```

```
drwxrwxr-x 2 boldi adm 4096 Oct 26 13:20 a
```

```
boldi@rivest:/tmp/test$ id
```

```
uid=1000(boldi) gid=4(adm) groups=4(adm),1000(boldi)
```

```
boldi@rivest:/tmp/test$ mkdir b
```

```
boldi@rivest:/tmp/test$ ls -la
```

```
total 344
```

```
drwxrwsr-x 4 boldi boldi 4096 Oct 26 13:23 .
```

```
drwxrwxrwt 3 root root 4096 Oct 26 13:23 ..
```

```
drwxrwxr-x 2 boldi adm 4096 Oct 26 13:20 a
```

```
drwxrwsr-x 2 boldi boldi 4096 Oct 26 13:23 b
```

In directories with group “s” (x+s) any new file will have the same group owner as the directory itself (and the “s” right is inhibited)

Example: Extra rights of the root user

```
root@shamir:/data/home/boldi/access_control# ls -la
total 28
```

```
drwxr-xr-x  3 boldi users  4096 2010-10-26 12:58 ./
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../
drwxr-xr-x  2 boldi users  4096 2010-10-26 12:58 test/
-rw-r--r--  1 boldi users    0 2010-10-26 12:58 test.txt
```

```
root@shamir:/data/home/boldi/access_control# id
```

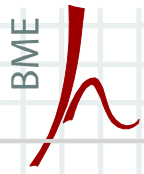
```
uid=0(root) gid=0(root) groups=0(root),124(smbusers),195(printusers)
```

```
root@shamir:/data/home/boldi/access_control# mkdir a
```

```
root@shamir:/data/home/boldi/access_control# ls -la
total 32
```

```
drwxr-xr-x  4 boldi users  4096 2010-10-26 13:30 ./
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../
drwxr-xr-x  2 root root    4096 2010-10-26 13:30 a/
drwxr-xr-x  2 boldi users  4096 2010-10-26 12:58 test/
-rw-r--r--  1 boldi users    0 2010-10-26 12:58 test.txt
```

Root used can write files in all directories, no matter of security settings



Root&g+s

```
boldi@shamir:~/access_control/f $ ls -la
```

```
total 20
```

```
drwxrwsr-x 2 boldi adm    4096 Apr  5 00:17 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 2 boldi eid      4 Apr  5 00:12 b
-rw-r--r-- 1 boldi users    6 Apr  5 00:04 c
-rw-r--r-- 2 boldi eid      4 Apr  5 00:12 d
```

```
boldi@shamir:~/access_control/f $ sudo touch e
```

```
boldi@shamir:~/access_control/f $ ls -la
```

```
total 20
```

```
drwxrwsr-x 2 boldi adm    4096 Apr  5 00:17 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 2 boldi eid      4 Apr  5 00:12 b
-rw-r--r-- 1 boldi users    6 Apr  5 00:04 c
-rw-r--r-- 2 boldi eid      4 Apr  5 00:12 d
-rw-r--r-- 1 root  adm      0 Apr  5 00:17 e
```

Expl: Root was able to make a new file in the directory, but that also inherited group ownership due to g+s.

Directory rights vs. file rights

```
boldi@shamir:~/access_control $ ls -la
```

```
total 24
```

```
drwxr-xr-x  2 boldi users 4096 2010-10-26 13:36 ./  
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../  
-rw-r--r--  1 root  root   0 2010-10-26 13:36 b
```

```
boldi@shamir:~/access_control $ id
```

```
uid=1005(boldi) gid=100(users) groups=30(dip),100(users)
```

```
boldi@shamir:~/access_control $ rm b
```

```
rm: remove write-protected regular empty file `b'? y
```

```
boldi@shamir:~/access_control $ ls -la
```

```
total 24
```

```
drwxr-xr-x  2 boldi users 4096 2010-10-26 13:36 ./  
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../  
boldi@shamir:~/access_control $
```

The file was owned by root, but the directory is writable by user boldi. Therefore he can delete the file.

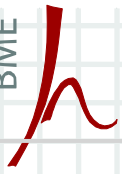
(he cannot write the file, but the deletion is basically “modifying the directory” in UNIX)

```

boldi@shamir:~/access_control/f $ ls -la
total 8
drwxr-xr-x 2 boldi users 4096 Apr  5 00:04 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 1 root  users    0 Apr  5 00:04 b
-rw-r--r-- 1 boldi users    6 Apr  5 00:04 c
boldi@shamir:~/access_control/f $ cp c b
cp: cannot create regular file `b': Permission denied
boldi@shamir:~/access_control/f $ rm b
rm: remove write-protected regular empty file `b'? y
boldi@shamir:~/access_control/f $ cp c b
boldi@shamir:~/access_control/f $ ls -la
total 16
drwxr-xr-x 2 boldi users 4096 Apr  5 00:05 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 1 boldi users    6 Apr  5 00:05 b
-rw-r--r-- 1 boldi users    6 Apr  5 00:04 c

```

The first copy command tries to overwrite the file rather than delete and re-make
 But manual remove and then copy works. Of course, in this case the file has new owner



- From the last slide we know that a simple copy on already existing file does not delete and re-make the file, so the rights might remain:

```
boldi@shamir:~/access_control/f $ ls -la
total 16
drwxr-xr-x 2 boldi users 4096 Apr  5 00:05 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-rw-rw- 1 root  users    2 Apr  5 00:08 b
-rw-r--r-- 1 boldi users    6 Apr  5 00:04 c
boldi@shamir:~/access_control/f $ cp c b
boldi@shamir:~/access_control/f $ ls -la
total 16
drwxr-xr-x 2 boldi users 4096 Apr  5 00:05 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-rw-rw- 1 root  users    6 Apr  5 00:08 b
-rw-r--r-- 1 boldi users    6 Apr  5 00:04 c
```

As You can see b is overwriten (see size), but the owner, rights remained.

What's up with group setuid+cp?

```
boldi@shamir:~/access_control/f $ ls -la
total 16
drwxrwsr-x 2 boldi adm    4096 Apr  5 00:05 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-rw-rw- 1 root  users    2 Apr  5 00:11 b
-rw-r--r-- 1 boldi users    6 Apr  5 00:04 c
boldi@shamir:~/access_control/f $ cp c b
boldi@shamir:~/access_control/f $ ls -la
total 16
drwxrwsr-x 2 boldi adm    4096 Apr  5 00:05 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-rw-rw- 1 root  users    6 Apr  5 00:11 b
-rw-r--r-- 1 boldi users    6 Apr  5 00:04 c
boldi@shamir:~/access_control/f $ rm b
boldi@shamir:~/access_control/f $ cp c b
boldi@shamir:~/access_control/f $ ls -la
total 16
drwxrwsr-x 2 boldi adm    4096 Apr  5 00:11 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 1 boldi adm     6 Apr  5 00:11 b
-rw-r--r-- 1 boldi users    6 Apr  5 00:04 c
```

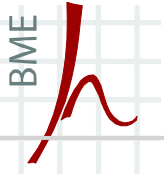
Explanation: Although there was groups setuid, cp did not make a new inode thus all rights remained as-is, but if we delete and remake file, that's different.

```
boldi@shamir:~/access_control/f $ ls -la
total 20
drwxrwsr-x 2 boldi users 4096 Apr  5 00:13 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 2 boldi users   4 Apr  5 00:12 b
-rw-r--r-- 1 boldi users   6 Apr  5 00:04 c
-rw-r--r-- 2 boldi users   4 Apr  5 00:12 d
boldi@shamir:~/access_control/f $ chgrp eid b
boldi@shamir:~/access_control/f $ ls -la
total 20
drwxrwsr-x 2 boldi users 4096 Apr  5 00:13 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 2 boldi eid    4 Apr  5 00:12 b
-rw-r--r-- 1 boldi users  6 Apr  5 00:04 c
-rw-r--r-- 2 boldi eid    4 Apr  5 00:12 d
```

Explanation: `chgrp` on `b` also affected rights of `d`, as `d` was a hard link to `b`.
(check 2 after the rights string).

Different behaviour of rm

```
boldi@shamir:~/access_control/f $ ls -la
total 20
drwxrwsr-x 2 boldi adm    4096 Apr  5 00:17 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 2 boldi eid     4 Apr  5 00:12 b
-rw-r--r-- 1 boldi users   6 Apr  5 00:04 c
-rw-r--r-- 2 boldi eid     4 Apr  5 00:12 d
-rw-r--r-- 1 root  adm     0 Apr  5 00:17 e
boldi@shamir:~/access_control/f $ rm e d c
rm: remove write-protected regular empty file `e'? y
boldi@shamir:~/access_control/f $
OS only asked about e, not the other files.
```



Rights on my own files

```
boldi@shamir:~/access_control $ touch c
boldi@shamir:~/access_control $ ls -la
total 24
drwxr-xr-x  2 boldi users 4096 2010-10-26 13:39 ./
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../
-rw-r--r--  1 boldi users   0 2010-10-26 13:39 c
boldi@shamir:~/access_control $ chmod a-r c
boldi@shamir:~/access_control $ ls -la
total 24
drwxr-xr-x  2 boldi users 4096 2010-10-26 13:39 ./
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../
--w-----  1 boldi users   0 2010-10-26 13:39 c
boldi@shamir:~/access_control $ cat c
cat: c: Permission denied
boldi@shamir:~/access_control $ chmod a+r c
boldi@shamir:~/access_control $ cat c
boldi@shamir:~/access_control $
```

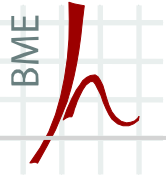
The read rights from file “c” are removed. Even if the owner is boldi, he is not permitted to read the file. But boldi is the owner of the file and thus give the access rights back. Now he can read the file.

(access right setting is allowed to root and the owner of the file)

- The above mentioned example does not relate to directory rights:

```
boldi@shamir:~/access_control/f $ ls -la
total 16
dr-xr-xr-x 2 boldi adm    4096 Apr  5 00:21 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 1 boldi eid      4 Apr  5 00:12 b
--w----- 1 boldi adm      4 Apr  5 00:21 c
boldi@shamir:~/access_control/f $ chmod a+r c
boldi@shamir:~/access_control/f $ ls -la
total 16
dr-xr-xr-x 2 boldi adm    4096 Apr  5 00:21 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 1 boldi eid      4 Apr  5 00:12 b
-rw-r--r-- 1 boldi adm      4 Apr  5 00:21 c
```

As You can see, the rights of e was recovered although there was no write permission on the directory, nor on the file.



Modifying access rights

```
boldi@shamir:~/access_control $ ls -la
total 24
drwxr-xr-x  2 boldi users 4096 2010-10-26 13:39 ./
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../
-rw-rw-r--  1 pek  users    0 2010-10-26 13:39 c
boldi@shamir:~/access_control $ chmod a-r c
chmod: changing permissions of `c': Operation not permitted
```

Even if the file is writable and the directory is writable, the access rights cannot be modified by other users than the owner

write r.&rename? Write->read r. ?

```
boldi@shamir:~/access_control $ ls -la
total 28
drwxr-xr-x  2 boldi users 4096 2010-10-26 13:51 ./
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../
-rw-rw-r--  1 pek  users   3 2010-10-26 13:51 c
boldi@shamir:~/access_control $ chmod a-r c
chmod: changing permissions of `c': Operation not permitted
boldi@shamir:~/access_control $ mv c d
boldi@shamir:~/access_control $ ls -la
total 28
drwxr-xr-x  2 boldi users 4096 2010-10-26 13:51 ./
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../
-rw-rw-r--  1 pek  users   3 2010-10-26 13:51 d
```

Boldi has no right to modify the access permissions of file 'c', but can rename it to 'd'. (as it is a directory write)

```
boldi@shamir:~/access_control $ ls -la
total 28
drwxr-xr-x  2 boldi users 4096 2010-10-26 13:53 ./
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../
--w--w----  1 boldi users   3 2010-10-26 13:51 e
boldi@shamir:~/access_control $ cat e
cat: e: Permission denied
```

Write permission does **not** mean that You can also read it.

Priorities among rights

```
boldi@shamir:~/access_control $ ls -la
total 28
drwxr-xr-x  2 boldi users 4096 2010-10-26 13:53 ./
drwxr-xr-x 189 boldi users 20480 2010-10-26 12:58 ../
----rw-r--  1 boldi users   3 2010-10-26 13:51 e
boldi@shamir:~/access_control $ cat e
cat: e: Permission denied
```

Even if others and group has read permission, the owner cannot read the file if he does not have it.

(The system only checks the owner's permissions if You are the owner, only check group permissions if You are not. The same stands for group and others.)

```
pek@shamir:/data/home/boldi/access_control$ cat e
Aa
```

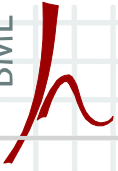
But other users (such as pek) can read the file...

- Some file systems, e.g. ext2, ext3 have extra possibilities, like file attributes (chattr,lsattr), “a” means append only

```
boldi@shamir:~/access_control/f $ ls -la
total 12
drwxrwxr-x 2 boldi adm    4096 Apr  5 00:32 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 1 boldi eid      4 Apr  5 00:12 b
-rw-r--r-- 1 boldi users    0 Apr  5 00:32 c
boldi@shamir:~/access_control/f $ sudo chattr +a c
boldi@shamir:~/access_control/f $ ls -la
total 12
drwxrwxr-x 2 boldi adm    4096 Apr  5 00:32 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 1 boldi eid      4 Apr  5 00:12 b
-rw-r--r-- 1 boldi users    0 Apr  5 00:32 c
boldi@shamir:~/access_control/f $ rm c
rm: cannot remove `c': Operation not permitted
```

```
boldi@shamir:~/access_control/f $ echo "aaaa" >c
-bash: c: Operation not permitted
boldi@shamir:~/access_control/f $ echo "aaaa" >>c
boldi@shamir:~/access_control/f $ ls -la
total 16
drwxrwxr-x 2 boldi adm    4096 Apr  5 00:32 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 1 boldi eid      4 Apr  5 00:12 b
-rw-r--r-- 1 boldi users    5 Apr  5 00:34 c
boldi@shamir:~/access_control/f $ echo "aaaa" >>c
boldi@shamir:~/access_control/f $ ls -la
total 16
drwxrwxr-x 2 boldi adm    4096 Apr  5 00:32 ./
drwxr-xr-x 3 boldi users 4096 Apr  5 00:04 ../
-rw-r--r-- 1 boldi eid      4 Apr  5 00:12 b
-rw-r--r-- 1 boldi users   10 Apr  5 00:34 c
boldi@shamir:~/access_control/f $ lsattr
-----a----- ./c
-----a----- ./b
```

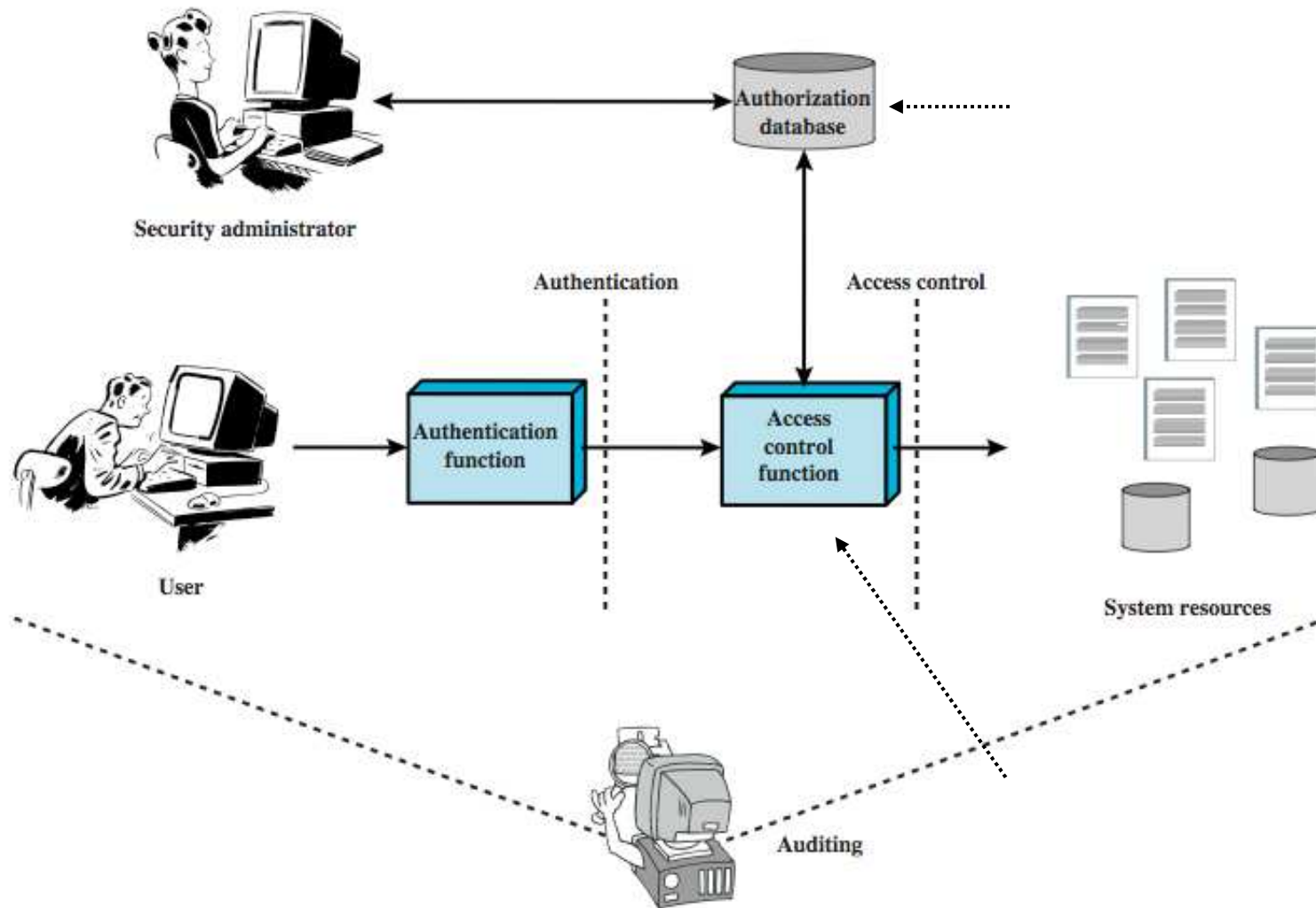
- The append right is useful e.g. In case of `.bash_history`: The shell can append new info to it, but the user won't be able to delete his history.

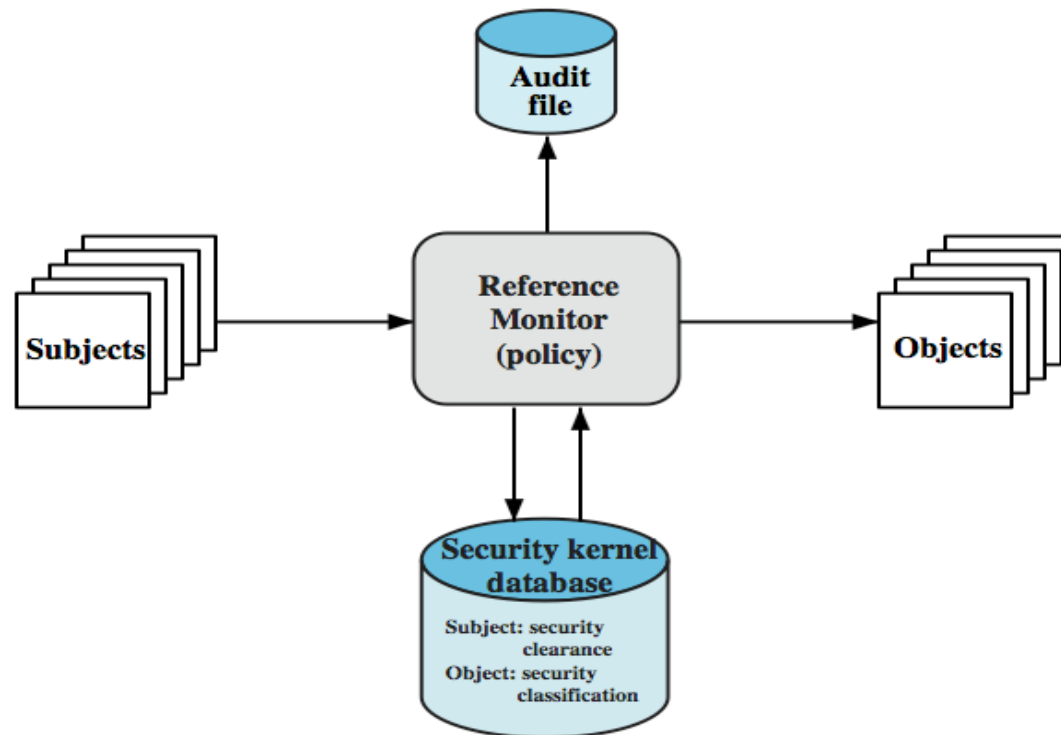


Access Control

- “The prevention of unauthorized use of a resource, including the prevention of use of a resource in an unauthorized manner“
- central element of computer security
- an access control *policy*
 - defines who (e.g., user, user group, process, etc.) can access what (system resources such as files, channels, services, etc.), in which manner (e.g., read, write, execute, etc.), and under what circumstances (e.g., time, location, history, etc.)
- access control *enforcement*
 - system components that ensure that the system operates in compliance with the access control policy
 - it should be impossible to circumvent the access control enforcement function
 - examples: firewalls, OS reference monitor

General Model of Access Control





- the reference monitor is a controlling element in the hardware and operating system of a computer that regulates the access of subjects to objects on the basis of security parameters of the subject and object

- **Discretionary access control (DAC)**
 - based on the identity of the requestor and on access rules (authorizations) stating what requestors are (or are not) allowed to do with the protected resources

- **Mandatory access control (MAC)**
 - based on comparing security **labels** (which indicate how sensitive or critical system resources are) with security **clearances** (which indicate system entities are eligible to access certain resources)

- **Role-based access control (RBAC)**
 - based on the roles that users have within the system and on rules stating what accesses are allowed to users in given roles

Basic Elements of AC

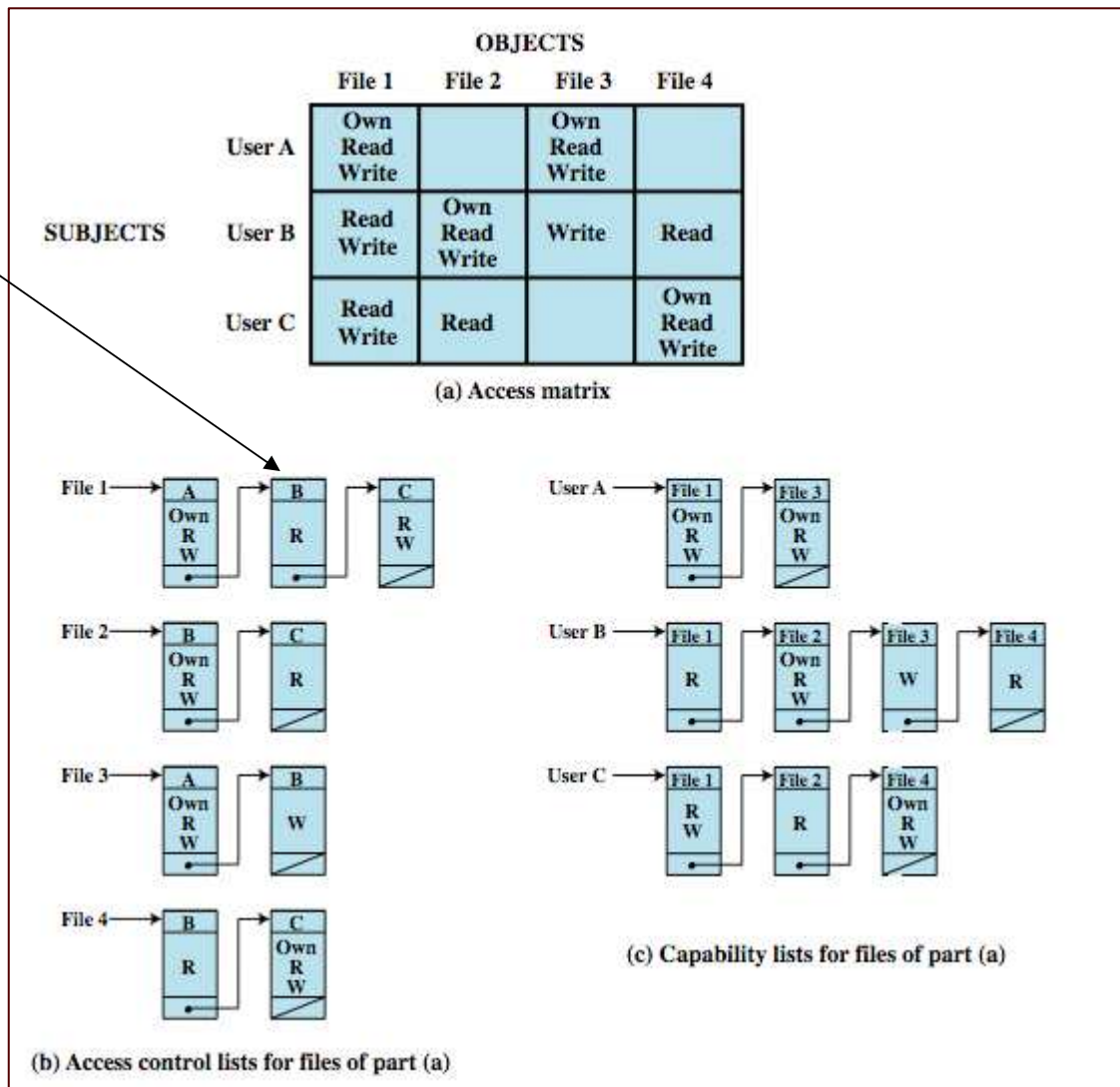
- **subject** - entity that can access objects
 - e.g., a user, user group, or a process representing a user
- **object** - access controlled resource
 - e.g. files, directories, records, programs, etc.
- **access right** - way in which subject accesses an object
 - e.g. read, write, execute, delete, create, search

Discretionary Access Control

- often represented in terms of an **access matrix**
 - lists subjects in one dimension (rows)
 - lists objects in the other dimension (columns)
 - each entry specifies access rights of the specified subject to that object
- the access matrix is often sparse
- can be decomposed by either row (credentials) or column (access control lists)
- another efficient representation is the authorization table, which contains (subject, object, access right) triplets
 - can be sorted (indexed) either by subject (→ credentials) or by objects (→ ACLs)

Decomposition of an AC Matrix

bug



A more general DAC model

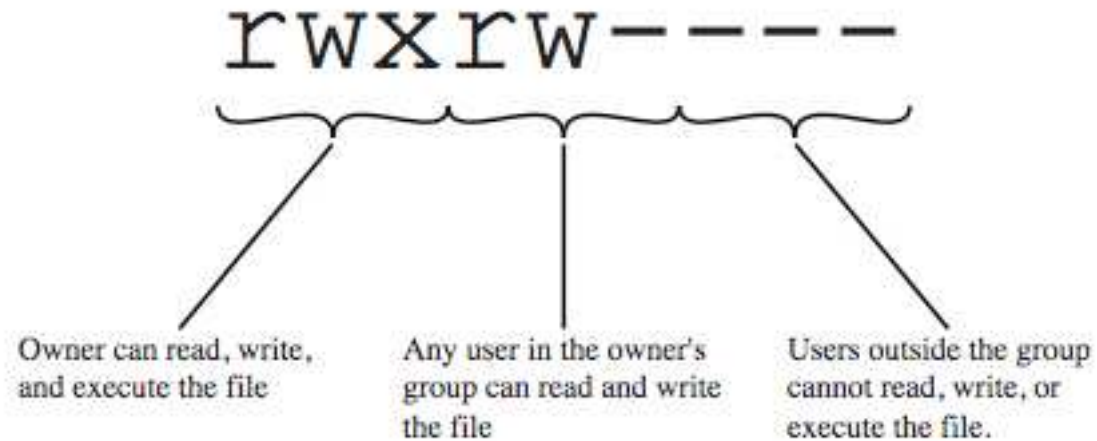
(Lampson, Graham, Denning)

- subjects are allowed to alter the protection state (represented by the AC matrix)
 - copy flag – transfer of the given access right to another subject (w/o copy flag)
 - owner – can grant any access right on the given object
 - control – can delete access rights assigned to the given subject

		OBJECTS								
		subjects			files		processes		disk drives	
		S ₁	S ₂	S ₃	F ₁	F ₁	P ₁	P ₂	D ₁	D ₂
SUBJECTS	S ₁	control	owner	owner control	read *	read owner	wakeup	wakeup	seek	owner
	S ₂		control		write *	execute			owner	seek *
	S ₃			control		write	stop			

* - copy flag set

Example: UNIX File Access Control



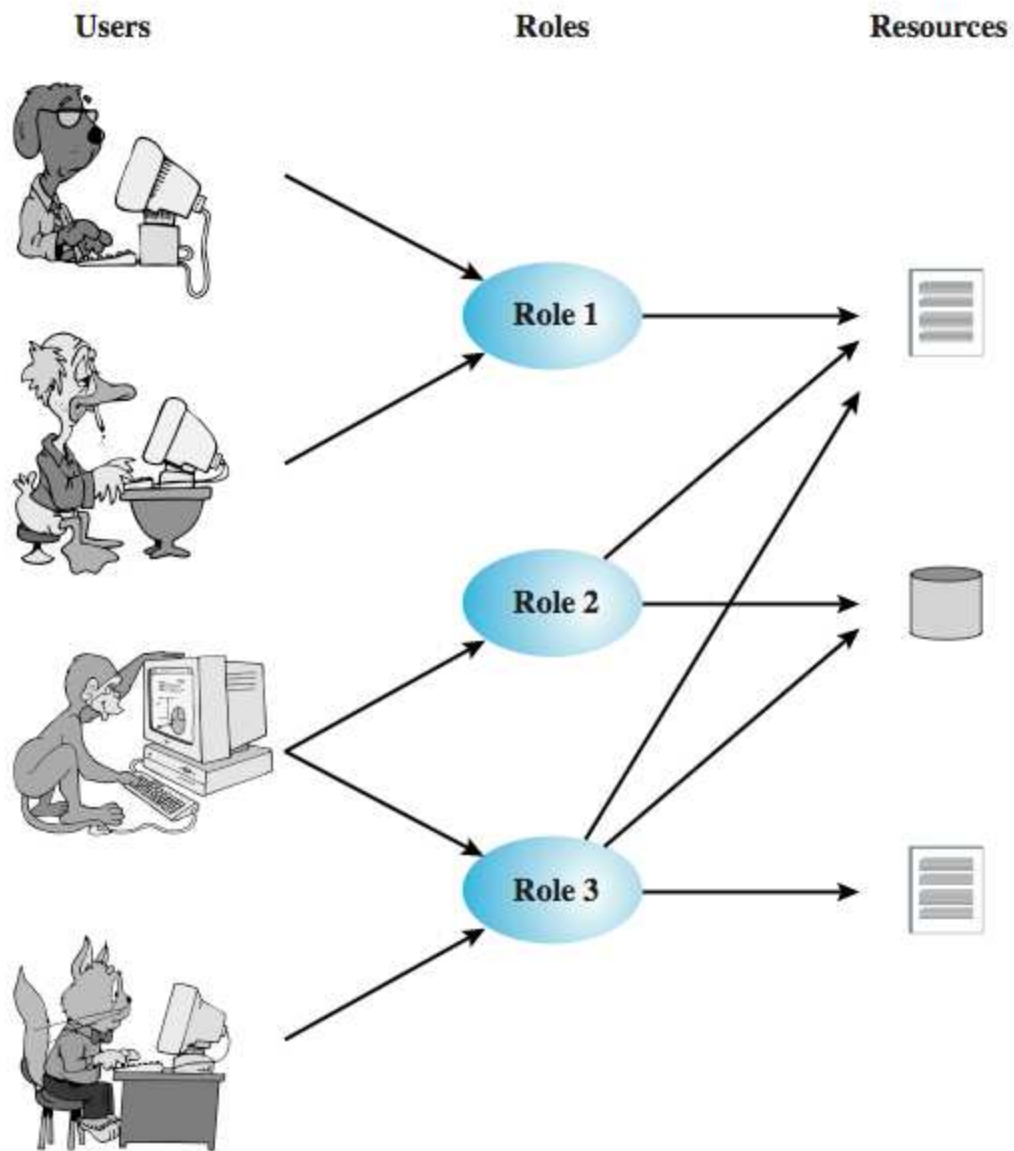
Example: UNIX File Access Control

- “set user ID”(SetUID) or “set group ID”(SetGID)
 - system temporarily uses rights of the file owner / group in addition to the real user’s rights when making access control decisions
 - enables privileged programs to access files / resources not generally accessible
- sticky bit
 - on directory limits rename/move/delete to owner - delete is not possible even if write right “+t”
 - (originally this bit causes the sytem to store the contents in memory as long as it is possible)
- superuser
 - is exempt from usual access control restrictions

Example: UNIX Access Control Lists

- modern UNIX systems support ACLs
- can specify any number of additional users / groups and associated rwx permissions
- ACLs are optional extensions to std perms
- group perms also set max ACL perms
- Important question: Multiple AC method is used, then: check all and allow if all accepts, or allow access if any accepts?
- when access is required
 - select most appropriate ACL
 - owner, named users, owning / named groups, others
 - check if have sufficient permissions for access

RBAC vs. DAC



RBAC vs. DAC

	R ₁	R ₂	...	R _n
U ₁	×			
U ₂	×			
U ₃		×		×
U ₄				×
U ₅				×
U ₆				×
•				
•				
U _m	×			

	OBJECTS								
	R ₁	R ₂	R _n	F ₁	F ₁	P ₁	P ₂	D ₁	D ₂
R ₁	control	owner	owner control	read *	read owner	wakeup	wakeup	seek	owner
R ₂		control		write *	execute			owner	seek *
•									
•									
R _n			control		write	stop			

- provides the means to reflect the hierarchical structure of roles in an organization
- make use of the concept of inheritance
 - (inheritance in ACL - directories?!)
- In Role Based methods:
 - a superior (child) role inherits all the rights of the subordinate (parent) role, and it may have more rights
 - may support multiple inheritance or limited to strict inheritance tree

Extensions: Constraints

- constraints define relationships among roles or conditions related to roles
- examples:
 - mutual exclusion:
 - a) set of roles such that a user can take only a single role in the set
 - b) set of roles such that any access right can be granted to only one role in the set
 - cardinality
 - max number of users who can take a role
 - max number of roles that can be taken (or activated for a given session) by a user
 - max number of roles that can have a given access right
 - prerequisite roles
 - a user can take a given role only if she already has some other prerequisite roles (strict hierarchies, least privilege principle)

Bell-LaPadula (BLP) Model

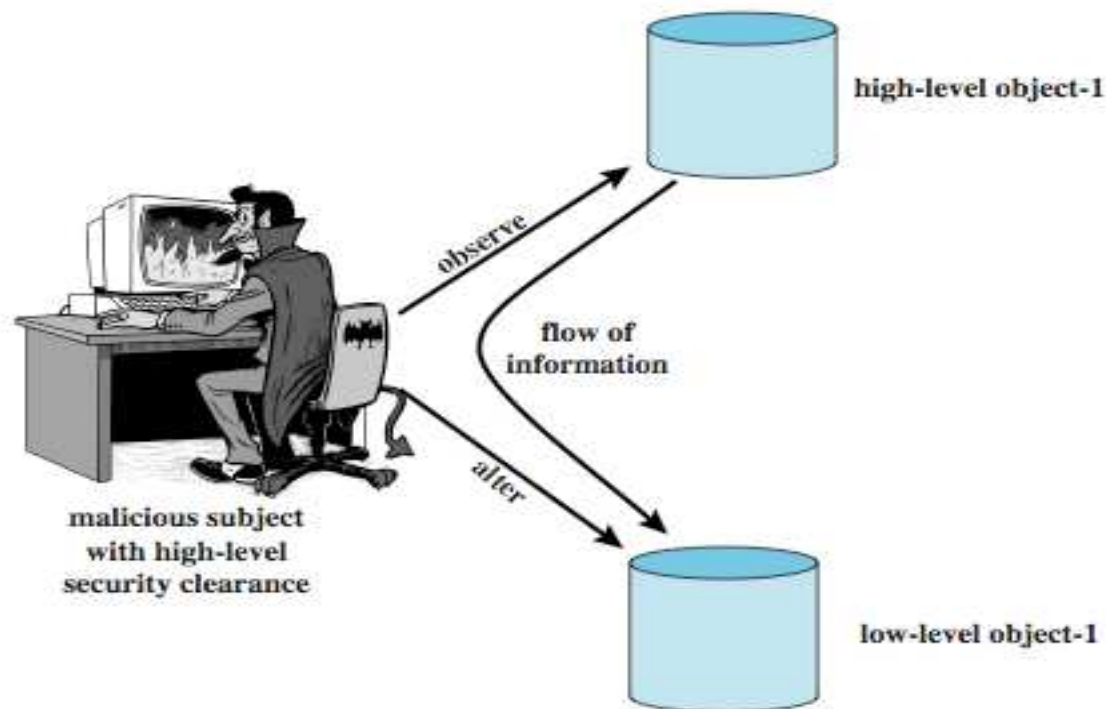
- developed in 1970s as a formal access control model (MAC + DAC)
- subjects and objects have a **security class**
 - top secret > secret > confidential > unclassified
 - subject has a **security clearance** level
 - object has a **security classification** level
 - these levels control how a subject may access an object

Requirements for Multi-Level Security

- **Multi-Level Security:**

prevent information flow from higher levels to lower levels

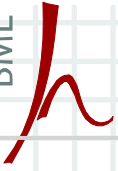
→no read up AND no write down (BLP)



Motivation for Formal Models

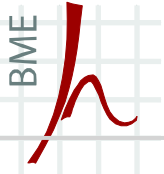
- two fundamental computer security facts:
 - all complex software systems have flaw/bugs
 - it is extremely difficult to build computer hardware/software not vulnerable to attacks
- hence desire to prove that the design and the implementation satisfy security requirements
- led to development of formal security models
 - initially funded by US DoD
- Bell-LaPadula (BLP) became very influential

- current state of system (b, M, f, H) :
 - b – current access set, triplets of form (subject, object, access mode)
 - M – Access Matrix, represents current set of permissions
 - f – **level function**, assigns security level to each object and subject
 - H – hierarchy, a directed rooted tree whose nodes are **objects** in the system, and the **security level** of an object dominates the security level of its parent



BLP operations

- 1. Get access:** Add a triple $(subject, object, access-mode)$ to the current access set b , used by a subject to initiate access to an object in the requested mode.
 - 2. Release access:** Remove a triple $(subject, object, access-mode)$ from the current access set b .
 - 3. Change object level:** Change the value of $f_o(O_j)$ for some object O_j . Used by a subject to alter the security level of an object.
 - 4. Change current level:** Change the value of $f_c(S_i)$ for some subject S_i . Used by a subject to alter the security level of a subject.
 - 5. Give access permission:** Add an access mode to some entry of the access permission matrix M . Used by a subject to grant an access mode on an object to another subject.
 - 6. Rescind access permission:** Delete an access mode from some entry of M . Used by a subject to revoke an access previously granted.
 - 7. Create an object:** Attach an object to the current tree structure H as a leaf. Used to create a new object.
 - 8. Delete a group of objects:** Detach from H an object and all other objects beneath it in the hierarchy. This operation may also modify the current access set b because all accesses to the object are released.
- Note: Rules 1 and 2 alter the current access; rules 3 and 4 alter the level functions; rules 5 and 6 alter access permission; and rules 7 and 8 alter the hierarchy.



BLP properties

- three BLP properties:
 - simple security (ss) property
 - *-property
 - discretionary security (ds) property

ss-property: for all (S_i, O_j, read) in b , $f_c(S_i) \geq f_o(O_j)$.

a subject at a given security level may not read an object at a higher security level

(no read-up).

***-property:** for all $(S_i, O_j, \text{append})$ in b , $f_c(S_i) \leq f_o(O_j)$ and
for all (S_i, O_j, write) in b , $f_c(S_i) = f_o(O_j)$

a subject at a given security level must not write to any object at a lower security level

(no write-down). The *-property is also known as the Confinement property.

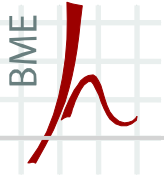
ds-property: for all (S_i, O_j, A_x) in b , $A_x \in M[S_i, O_j]$

use of an [access matrix](#) to specify the discretionary access control.

An individual may grant to another individual access to a file based on the owner's discretion constrained by the MAC rules.

Definition and proof of security

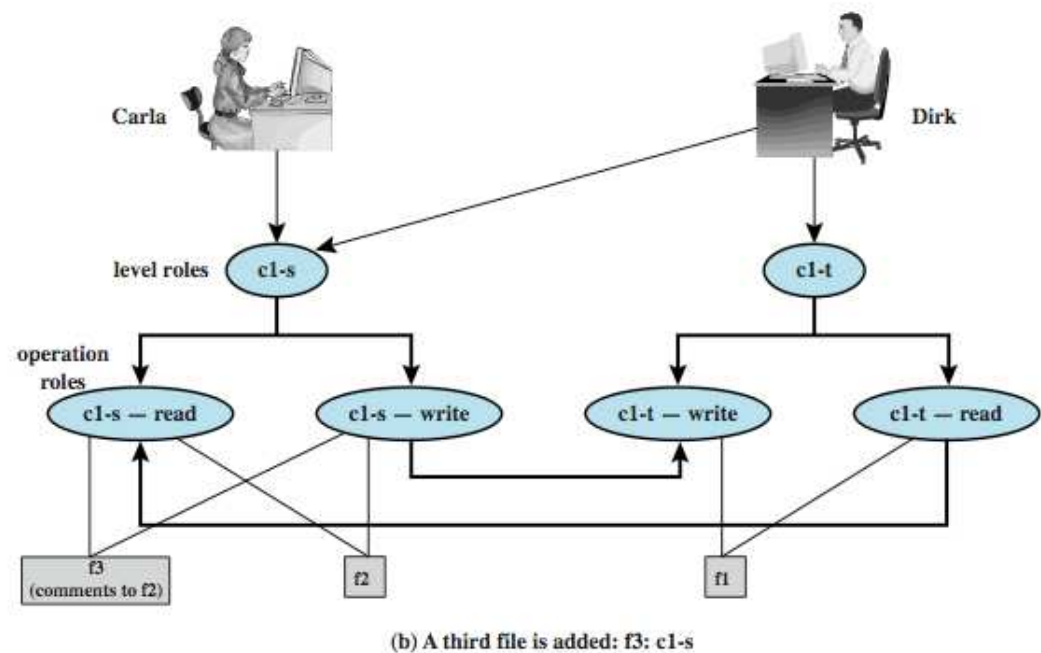
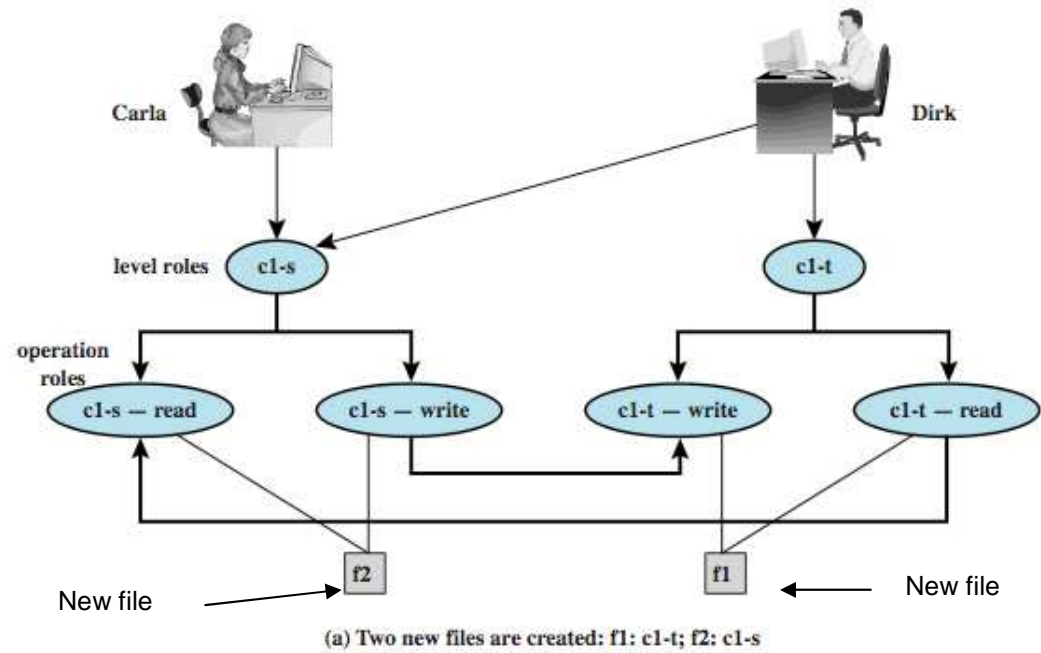
- The three BLP properties can be used to define a secure system as:
 1. initial state of the system satisfies the BLP properties
 2. every state change preserves the BLP properties
- in theory, it is possible to prove the system is secure (above properties are satisfied)
- in practice, for large systems, such proofs are usually not feasible



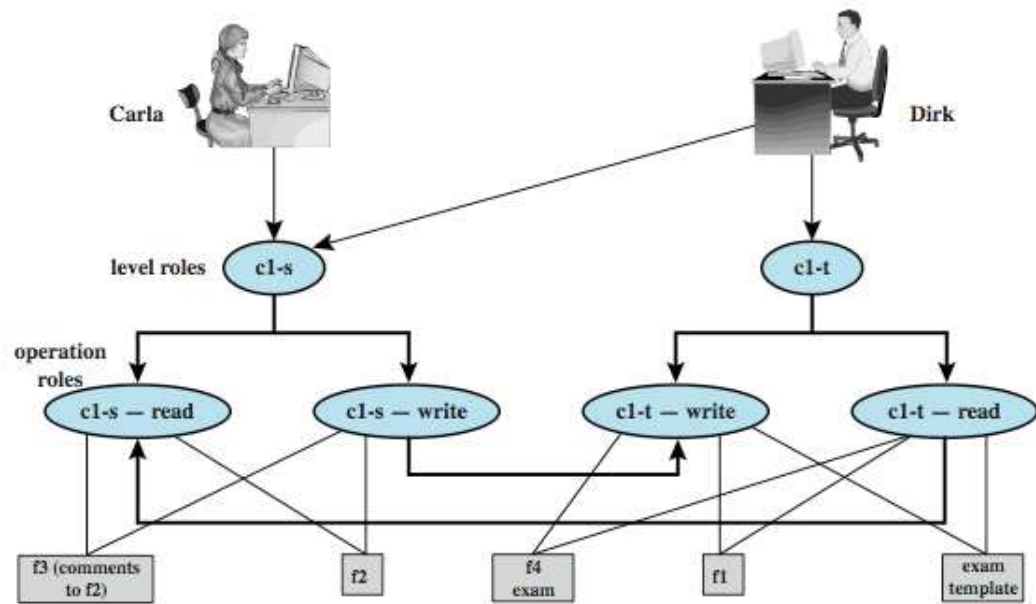
BLP example

- Carla – student, Dirk – teacher
- Carla – 1 role
- Denoted by C1-s
- Dirk: two access permissions (roles)
- Denoted by C1-t, c1-s

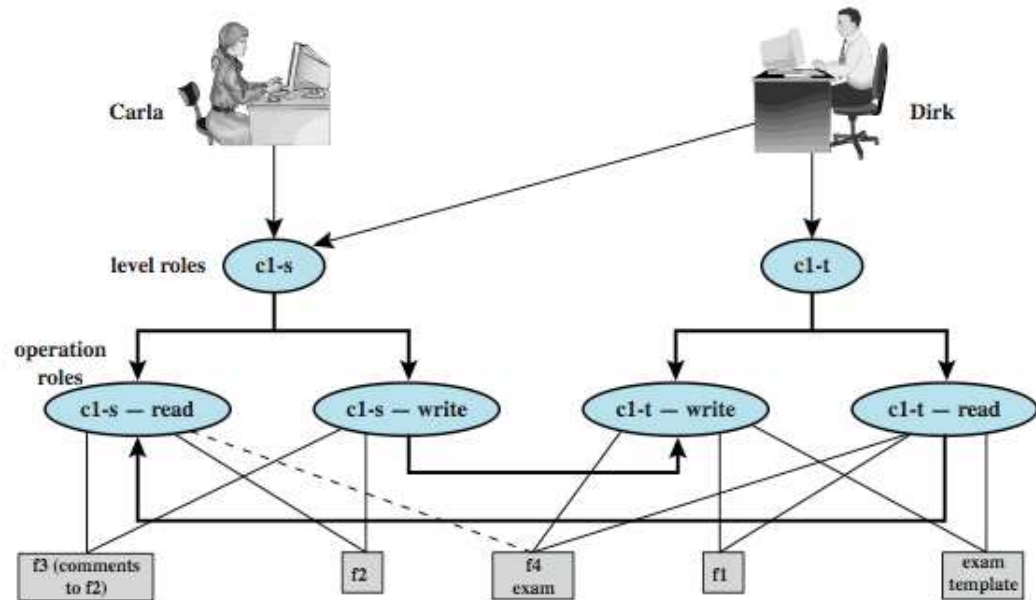
BLP Example



BLP Example cont.

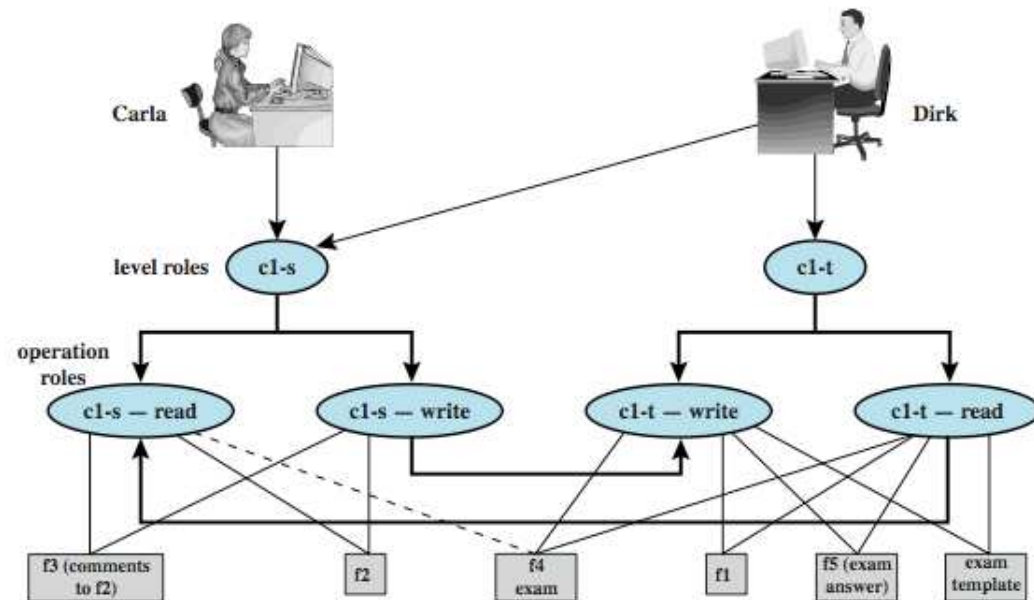


(c) An exam is created based on an existing template: f4: c1-t

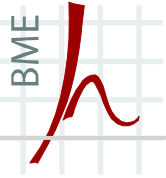


(d) Carla, as student, is permitted access to the exam: f4: c1-s

BLP Example cont.



(e) The answers given by Carla are only accessible for the teacher: f5: c1-t



Biba Integrity Model

BLP deals only with **confidentiality**

- **Biba** is a similar model for **integrity**
- operations are not much different:
 - modify – write or update an object
 - observe – read an object
 - invoke – communication between subjects

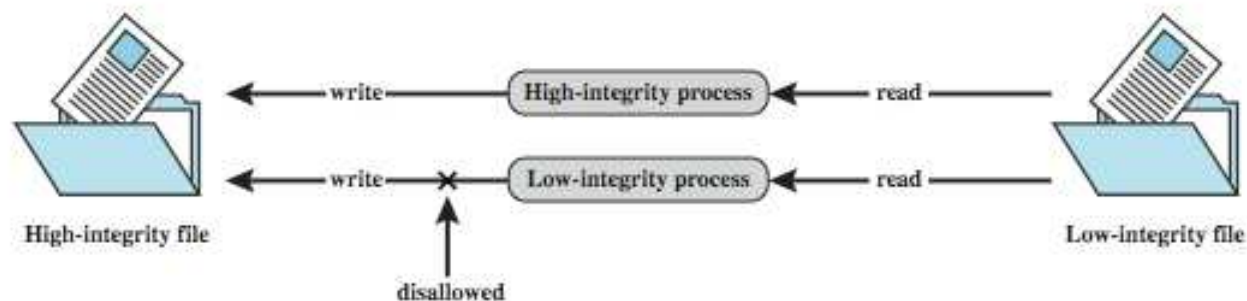
Biba Integrity Model

- Biba integrity properties:

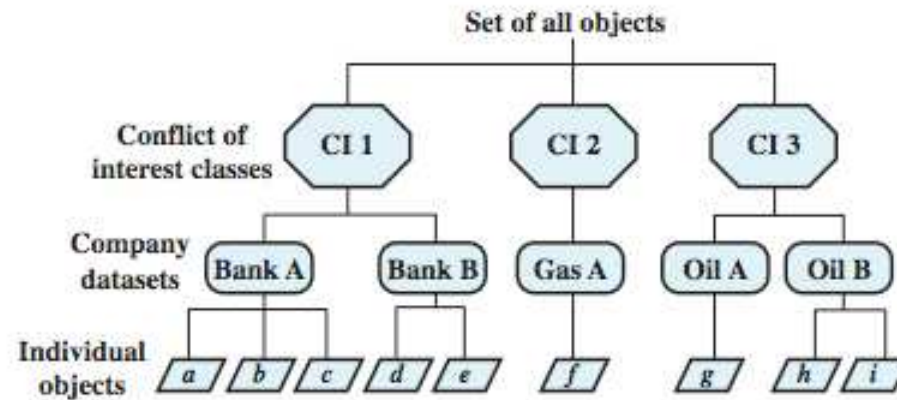
Simple integrity: A subject can modify an object only if the integrity level of the **subject** dominates the integrity level of the **object**: $I(S) \geq I(O)$.

Integrity confinement: A subject can read on object only if the integrity level of the subject is dominated by the integrity level of the **object**: $I(S) \leq I(O)$.

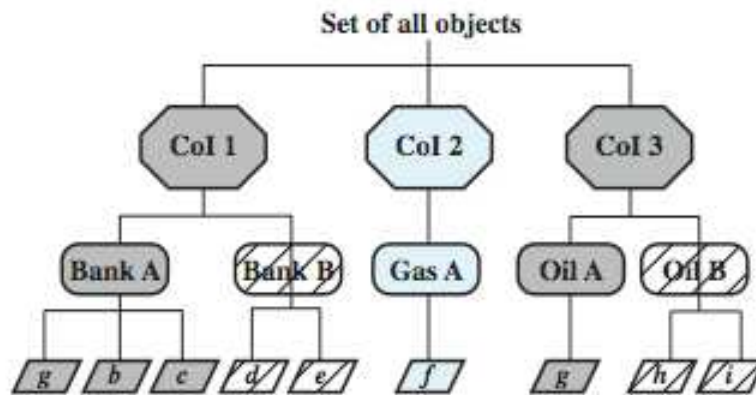
Invocation property: A subject can invoke another subject only if the integrity level of the 1st subject dominates the integrity level of the 2nd subject: $I(S_1) \geq I(S_2)$.



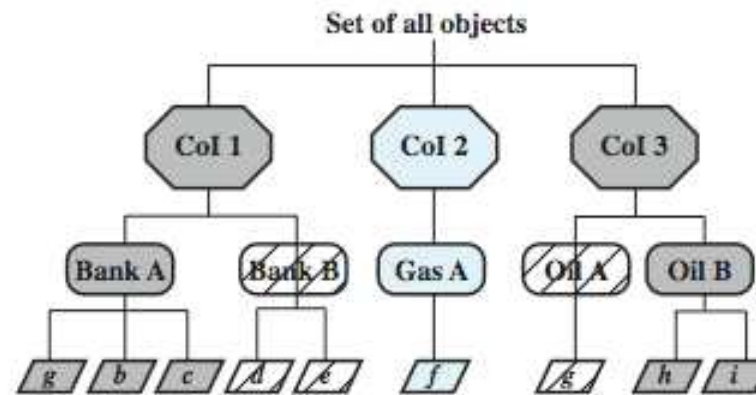
Chinese Wall Model



(a) Example set



(b) John has access to Bank A and Oil A



(c) Jane has access to Bank A and Oil B

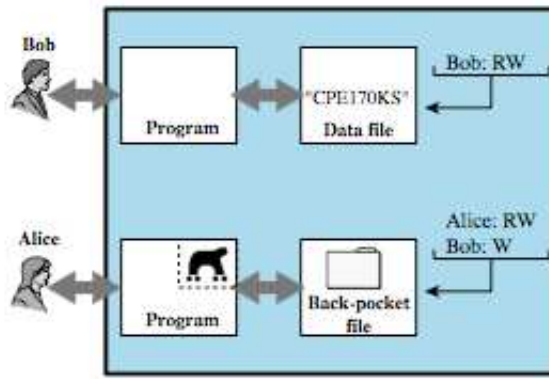
Chinese Wall Model

- CWM rules:

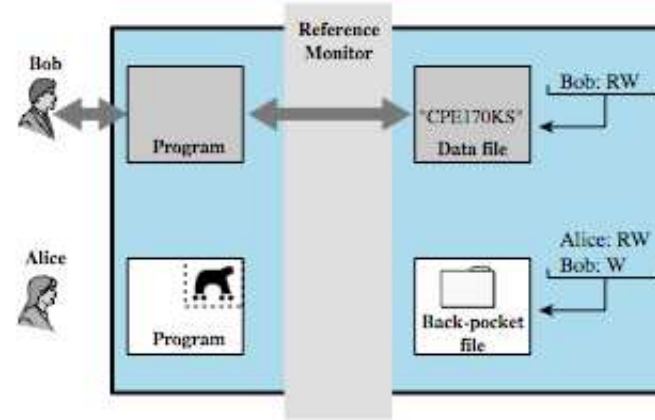
Simple security rule: A subject S can read on object O only if: O is in the same DS as an object already accessed by S , **or** O belongs to a CI from which S has not yet accessed any information.

***-property rule:** A subject S can write an object O only if: S can read O according to the simple security rule, **and** all objects that S can read are in the same DS as O .

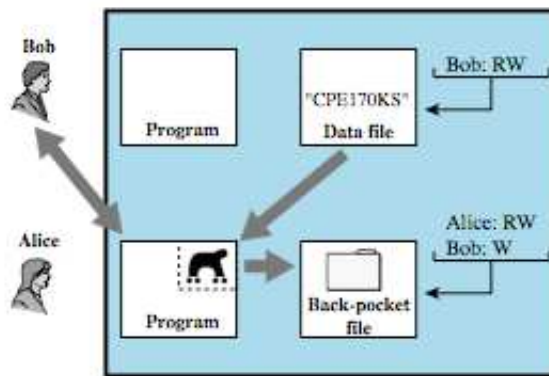
Example: Trojan Horse Defense



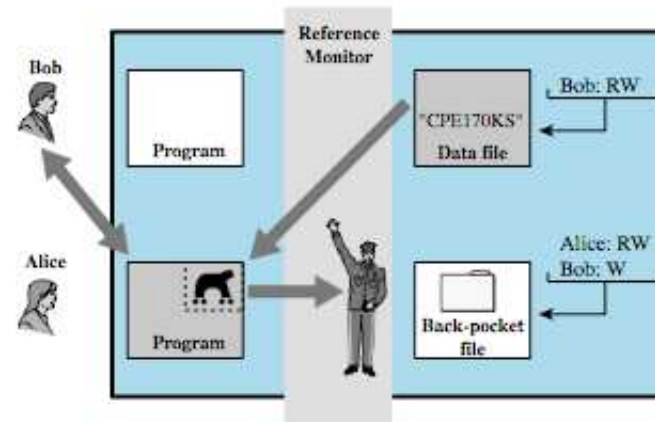
(a)



(c)



(b)



(d)

Example:MLS Database Security

Department Table - U		
Did	Name	Mgr
4	accts	Cathy
8	PR	James

Employee - R			
Name	Did	Salary	Eid
Andy	4	43K	2345
Calvin	4	35K	5088
Cathy	4	48K	7712
James	8	55K	9664
Ziggy	8	67K	3054

(a) Classified by table

Department Table		
Did -U	Name -U	Mgr -R
4	accts	Cathy
8	PR	James

Employee			
Name -U	Did -U	Salary -R	Eid -U
Andy	4	43K	2345
Calvin	4	35K	5088
Cathy	4	48K	7712
James	8	55K	9664
Ziggy	8	67K	3054

(b) Classified by column (attribute)

Example: ML Database Security

Department Table			
Did	Name	Mgr	
4	accts	Cathy	R
8	PR	James	U

Employee				
Name	Did	Salary	Eid	
Andy	4	43K	2345	U
Calvin	4	35K	5088	U
Cathy	4	48K	7712	U
James	8	55K	9664	R
Ziggy	8	67K	3054	R

(c) Classified by row (tuple)

Department Table		
Did	Name	Mgr
4 - U	accts - U	Cathy - R
8 - U	PR - U	James - R

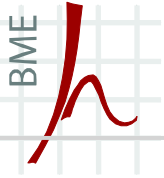
Employee			
Name	Did	Salary	Eid
Andy - U	4 - U	43K - U	2345 - U
Calvin - U	4 - U	35K - U	5088 - U
Cathy - U	4 - U	48K - U	7712 - U
James - U	8 - U	55K - R	9664 - U
Ziggy - U	8 - U	67K - R	3054 - U

(d) Classified by element

MLS Database Security

Read Access

- DBMS enforces simple security rule (no read up)
 - easy if granularity entire database / table level
 - inference problems if have column granularity
 - if can query on restricted data can infer its existence
 - `SELECT Ename FROM Employee WHERE Salary > 50K`
 - solution is to check access to all query data
- The system must prevent not just a read up but a query that must access higher-level elements in order to determine the response!



MLS Database Security

Write Access

- enforce *-security rule (no write down)
 - have problem if a low clearance user wants to insert a row with a primary key that already exists in a higher level row:
 - can reject, but user knows row exists
 - can replace, compromises data integrity
 - can use polyinstantiation and insert multiple rows with same key, creates conflicting entries
 - same alternatives occur on update
- Avoid problem by using database / table granularity!

Mandatory Access Control in practice

- Linux uses a DAC security model
- but Mandatory Access Controls (MAC) impose a global security policy on all users
 - users may not set controls weaker than policy
 - normal admin done with accounts without authority to change the global security policy
 - but MAC systems have been hard to manage
- **RSBAC** (www.rsbac.de) for Linux
- Novell's SuSE Linux has **AppArmor**
- RedHat Enterprise Linux has **SELinux**
- pure SELinux for high-sensitivity, high-security

- is NSA's powerful implementation of mandatory access controls for Linux
- Linux DACs still applies, but if it allows the action SELinux then evaluates it against its own security policies
- to manage complexity SELinux has:
 - "which is not expressly permitted, is denied"
 - groups of subjects, permissions, and objects

- have **Role Based Access Control (RBAC)**
 - rules specify **roles** a user may assume
 - other rules specify circumstances when a user may **transition** from one role to another
- and **Multi Level Security (MLS)** (with basics from BLP model)
 - concerns handling of classified data
 - “no read up, no write down”
 - MLS is enforced via file system labeling
 - Check http://selinuxproject.org/page/NB_MLS for more info

- RSBAC is a security extension for the Linux security model
- Kernel space modifications and user space management tools
- Multiple security models supported, MAC, ACL, Role Based models and specialities: Dazuko antivirus interface for access control, File Flags, Jail etc.
- Can disable standard Linux DAC, or work simultaneously
- One of the most interesting/usable solution is the Role Based Access Control

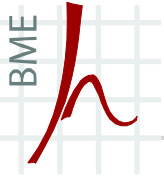
- File types:
- Executables / shell
- Libraries
- Web HTML files
- CGI scripts
- FTP data files
- Shell User files
- Configuration files
- Database files
- Temporary files
- Special files (sockets, devices, etc.)
- Password files and other files with secret information
- Email boxes, temporary email files (queue) etc.
- Log files
- Package management related files (downloaded packages, installation information)
- Help files (Man pages, etc.)

Example / Roles for RBAC in linux

- General User
- Security Officer
- System administrator – file and process admin
- Installer
- Database handler (server)
- Web server
- FTP server/user
- SMTP server
- Log file handler
- ... Other server processes, e.g cron, etc.
- ...

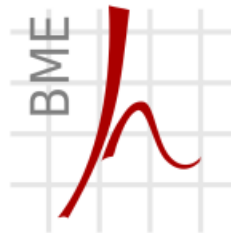
Example / access rights

- The web server can read(write?) HTML files, can run CGI scripts, but cannot read User data files, FTP data files, cannot run shell, cannot read log files, ...
- The database server can read and write database files, and can communicate by sockets, but cannot run any other executables, cannot read/write other files than its own executables and data files,...



Kérdések?

KÖSZÖNÖM A FIGYELMET!



Híradástechnikai Tanszék

Dr. Bencsáth Boldizsár
adjunktus
BME Híradástechnikai Tanszék
bencsath@crysys.hit.bme.hu

