

# NETWORK SECURITY PRACTICES – ATTACK AND DEFENSE

---

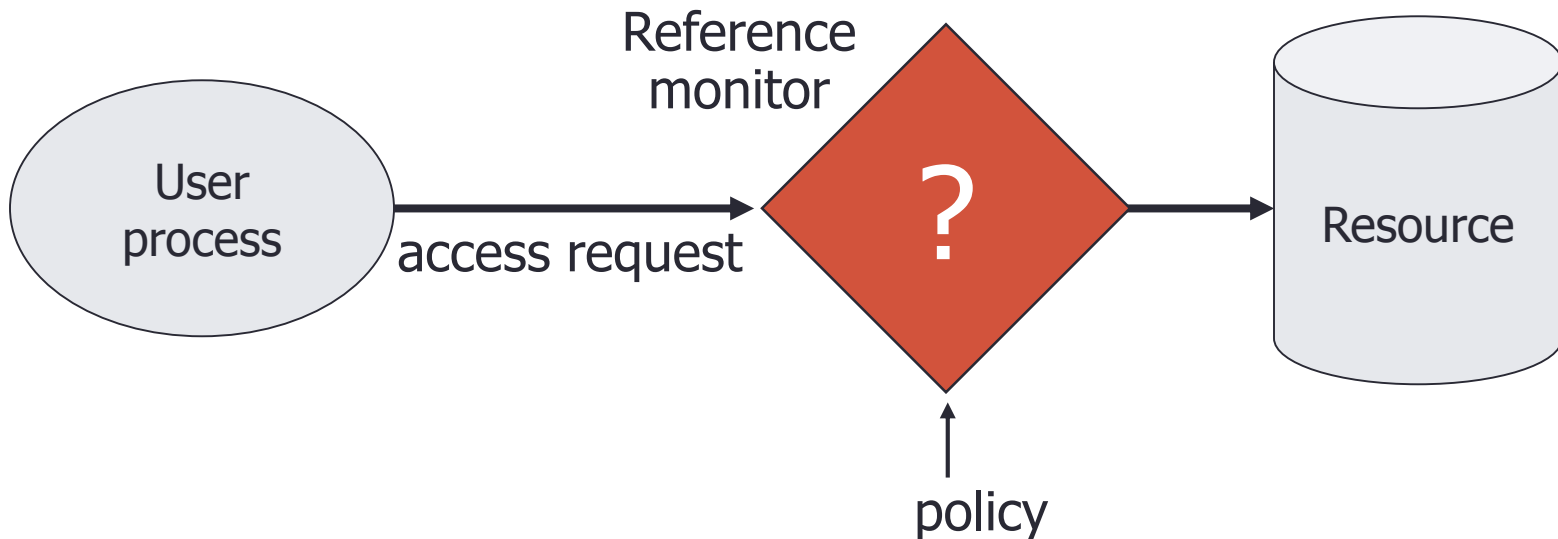
Unix Access Control

# Roadmap

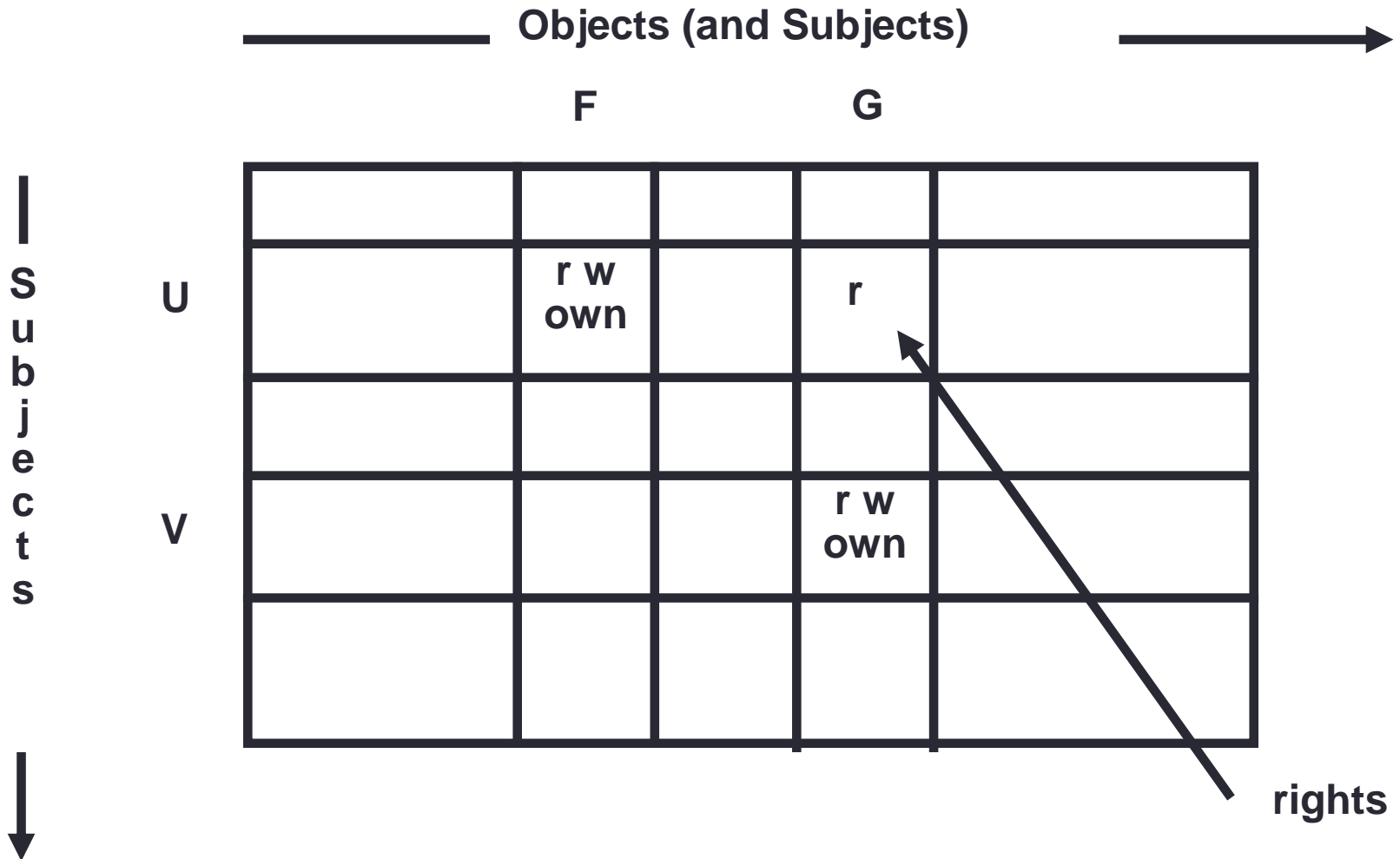
- Basic Concepts in Access Control
- UNIX Access Control Overview
- Files in UNIX
- Processes in UNIX

# Access control

- A **reference monitor** mediates all access to resources
  - Tamper-proof:
  - **Complete mediation**: control **all** accesses to resources
  - Small enough to be analyzable



# ACCESS MATRIX MODEL



# ACCESS MATRIX MODEL

- Basic Abstractions
  - Subjects
  - Objects
  - Rights
- The rights in a cell specify the access of the subject (row) to the object (column)

# PRINCIPALS AND SUBJECTS

- A subject is a program (application) executing on behalf of some principal(s)
- A principal may at any time be idle, or have one or more subjects executing on its behalf

**What are subjects in UNIX?**

**What are principals in UNIX?**

# OBJECTS

- An object is anything on which a subject can perform operations (mediated by rights)
- Usually objects are passive, for example:
  - File
  - Directory (or Folder)
  - Memory segment
- But, subjects can also be objects, with operations
  - kill
  - suspend
  - resume

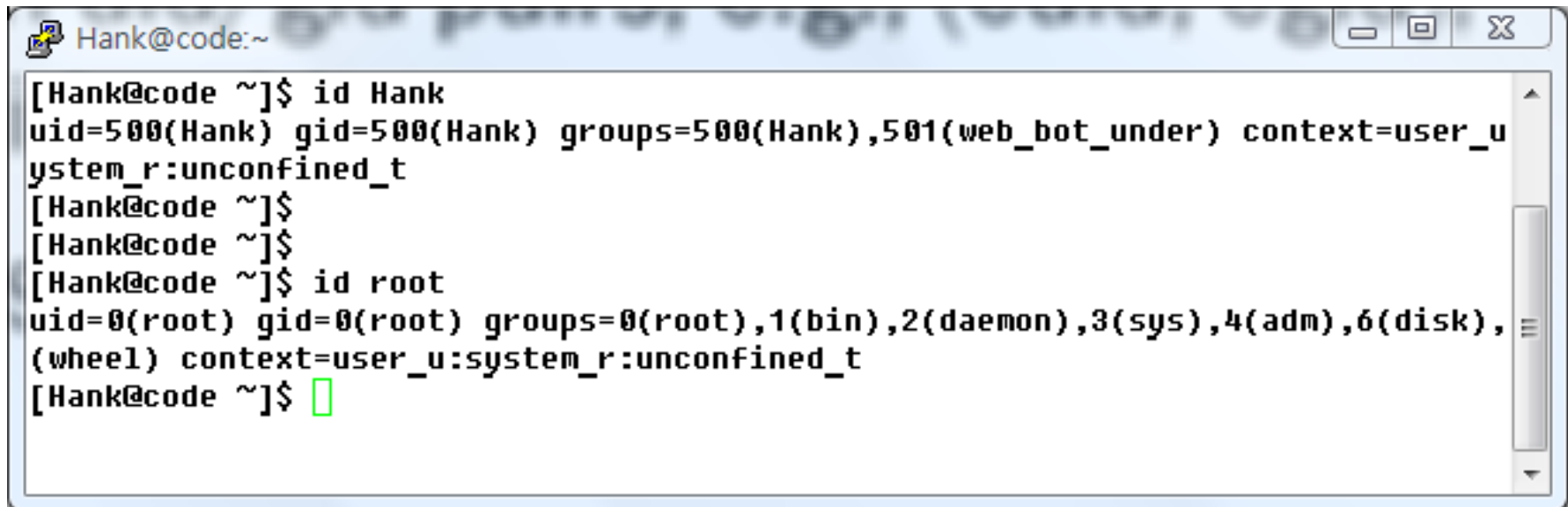
# Roadmap

- Basic Concepts in Access Control
- UNIX Access Control Overview
- Files in UNIX
- Processes in UNIX



# Basic Concepts of UNIX Access Control: Users, Groups, Files, Processes

- Each user account has a unique UID
  - UID=0 for the super user (root)
- A user account belongs to multiple groups



```
Hank@code:~  
[Hank@code ~]$ id Hank  
uid=500(Hank) gid=500(Hank) groups=500(Hank),501(web_bot_under) context=user_u  
system_r:unconfined_t  
[Hank@code ~]$  
[Hank@code ~]$  
[Hank@code ~]$ id root  
uid=0(root) gid=0(root) groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),  
(wheel) context=user_u:system_r:unconfined_t  
[Hank@code ~]$
```

# Basic Concepts of UNIX Access Control: Users, Groups, Files, Processes

- Subjects are processes
  - associated with uid/gid pairs, e.g., (euid, egid), (ruid, rgid), (suid, sgid)
- Objects are files

# Roadmap

- Basic Concepts in Access Control
- UNIX Access Control Overview
- Files in UNIX
- Processes in UNIX

# Organization of Objects

- Almost all objects are modeled as files
  - Files are arranged in a hierarchy
  - Files exist in directories
  - Directories are also one kind of files
- Each object has
  - owner
  - group
  - 12 permission bits
    - rwx for owner, rwx for group, and rwx for others
    - suid, sgid, sticky

# inode

```
[Hank@sense 2010_nspad]$ ls -l
total 96
drwxr-xr-x 3 root root 4096 May  3 22:16 2010_nspad_files
-rw-r--r-- 1 root root 71572 May 10 15:53 2010_nspad.htm
drwxr-xr-x 3 root root 4096 May 10 15:52 slides
[Hank@sense 2010_nspad]$
[Hank@sense 2010_nspad]$ ls -li
2456038 2010_nspad_files 2456037 2010_nspad.htm 2455983 slides
[Hank@sense 2010_nspad]$
[Hank@sense 2010_nspad]$ stat 2010_nspad.htm
  File: `2010_nspad.htm'
  Size: 71572          Blocks: 160          IO Block: 4096   regular file
Device: fd00h/64768d  Inode: 2456037       Links: 1
Access: (0644/-rw-r--r--)  Uid: (   0/   root)   Gid: (   0/   root)
Access: 2010-05-12 22:48:00.000000000 +0800
Modify: 2010-05-10 15:53:20.000000000 +0800
Change: 2010-05-10 15:53:20.000000000 +0800
[Hank@sense 2010_nspad]$ █
```

# inode

- A representation of a file and its metadata (timestamps, type, size, attributes)
  - It does not contain the file name
  - It does not include the data stored in the file
- inodes can represent files, directories, symbolic links, and special files
- inode metadata (uid owner, mode, file timestamps, etc.)

```

/*
 * Structure of an inode on the disk
 */
struct ext3_inode {
    __le16 i_mode;      /* File mode */
    __le16 i_uid;       /* Low 16 bits of Owner Uid */
    __le32 i_size;      /* Size in bytes */
    __le32 i_atime;     /* Access time */
    __le32 i_ctime;     /* Creation time */
    __le32 i_mtime;     /* Modification time */
    __le32 i_dtime;     /* Deletion Time */
    __le16 i_gid;       /* Low 16 bits of Group Id */
    __le16 i_links_count; /* Links count */
    __le32 i_blocks;    /* Blocks count */
    __le32 i_flags;     /* File flags */
    union {
        struct {
            __u32 l_i_reserved1;
        } linux1;
        struct {
            __u32 h_i_translator;
        } hurd1;
        struct {
            __u32 m_i_reserved1;
        } masix1;
    } osd1;             /* OS dependent 1 */
    __le32 i_block[EXT3_N_BLOCKS]; /* Pointers to blocks */
    __le32 i_generation; /* File version (for NFS) */
    __le32 i_file_acl; /* File ACL */
    __le32 i_dir_acl; /* Directory ACL */
    __le32 i_faddr; /* Fragment address */

```

```

Hank@Ruby:~/tmp

/* Tell code we have these members. */
#define _STATBUF_ST_BLKSIZE
#define _STATBUF_ST_RDEV
/* Nanosecond resolution time values are supported. */
#define _STATBUF_ST_NSEC

/* Encoding of the file mode. */

#define __S_IFMT          0170000 /* These bits determine file type. */

/* File types. */
#define __S_IFDIR          0040000 /* Directory. */
#define __S_IFCHR          0020000 /* Character device. */
#define __S_IFBLK          0060000 /* Block device. */
#define __S_IFREG          0100000 /* Regular file. */
#define __S_IFIFO          0010000 /* FIFO. */
#define __S_IFLNK          0120000 /* Symbolic link. */
#define __S_IFSOCK          0140000 /* Socket. */

/* POSIX.1b objects. Note that these macros always evaluate to zero. But
   they do it by enforcing the correct use of the macros. */
#define __S_TYPEISMQ(buf) ((buf)->st_mode - (buf)->st_mode)
#define __S_TYPEISSEM(buf) ((buf)->st_mode - (buf)->st_mode)
#define __S_TYPEISSHM(buf) ((buf)->st_mode - (buf)->st_mode)

/* Protection bits. */

#define __S_ISUID          04000 /* Set user ID on execution. */
#define __S_ISGID          02000 /* Set group ID on execution. */
#define __S_ISVTX          01000 /* Save swapped text after use (sticky). */
#define __S_IREAD          0400  /* Read by owner. */
#define __S_IWRITE         0200  /* Write by owner. */
#define __S_IEXEC          0100  /* Execute by owner. */

203,47-56 Bot

```

(/usr/include/bits/stat.h)

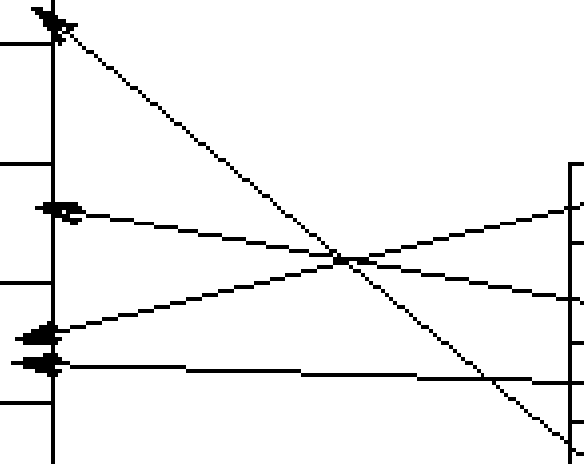


# UNIX Directories

Inode table


Directory

i1	name1
i2	name2
i1	name3
i4	name4



# Basic Permissions Bits on Files (Non-directories)

- Read controls reading the content of a file
  - i.e., the read system call
- Write controls changing the content of a file
  - i.e., the write system call
- Execute controls loading the file in memory and execute
  - i.e., the execve system call

# Execution of a file

- Binary file vs. Script file
- Having execute but not read, can one run a binary file?
- Having execute but not read, can one run a script file?
  - No.
- Having read but not execute, can one run a script file?
  - Yes, by invoking the interpreter

# Permission Bits on Directories

- A directory is a special file on Unix filesystem
- Read bit allows one to show file names in a directory
- Execution bit controls traversing a directory
  - does a lookup, allows one to find inode # from file name
  - `chdir` to a directory requires execution
- Write + execution control creating/deleting files in the directory
  - Deleting a file under a directory requires no permission on the file
- Accessing a file identified by a path name requires execution to all directories along the path

# The suid, sgid, sticky bits

	suid	sgid	sticky bit
non-executable files	no effect	affect locking (unimportant for us)	not used anymore
executable files	change euid when executing the file	change egid when executing the file	not used anymore
directories	no effect	new files inherit group of the directory	only the owner of a file can delete

# Some Examples

- What permissions are needed to access a file/directory?
  - read a file: /d1/d2/f3
  - write a file: /d1/d2/f3
  - delete a file: /d1/d2/f3
  - rename a file: from /d1/d2/f3 to /d1/d2/f4
  - ...
- File/Directory Access Control is by System Calls
  - e.g., open(2), stat(2), read(2), write(2), chmod(2), opendir(2), readdir(2), readlink(2), chdir(2), ...

# The Three sets of permission bits

- Intuition:
  - if the user is the owner of a file, then the r/w/x bits for owner apply
  - otherwise, if the user belongs to the group the file belongs to, then the r/w/x bits for group apply
  - otherwise, the r/w/x bits for others apply
- Can one implement negative authorization, i.e., only members of a particular group are not allowed to access a file?

# Other Issues On Objects in UNIX

- Accesses other than read/write/execute
  - Who can change the permission bits?
    - The owner can
  - Who can change the owner?
    - Only the superuser
- Rights not related to a file
  - Affecting another process
  - Operations such as shutting down the system, mounting a new file system, listening on a low port
    - traditionally reserved for the root user



# Roadmap

- Basic Concepts in Access Control
- UNIX Access Control Overview
- Files in UNIX
- Processes in UNIX

# Subjects vs. Principals

- Access rights are specified for users (accounts)
- Accesses are performed by processes (subjects)
- The OS needs to know on which users' behalf a process is executing

# Process User ID Model in Modern UNIX Systems

- Each process has three user IDs
  - real user ID (ruid)                      owner of the process
  - effective user ID (euid)                used in most access control decisions
  - saved user ID (suid)
- and three group IDs
  - real group ID
  - effective group ID
  - saved group ID

# Process User ID Model in Modern UNIX Systems

The image shows two terminal windows. The top window, titled 'Hank@sense:~', shows a user login and password change process. The bottom window, titled 'root@sense:/proc/26392', shows the output of the command 'cat /proc/26392/status'. The output lists various process attributes, including user IDs. Annotations highlight the 'effective', 'real', and 'saved' user IDs.

```
login as: Hank
Hank@sense.cs.nctu.edu.tw's password:
Last login: Sun May 16 23:26:45 2010 from 114-37-137-210.dynamic.hinet.net
[Hank@sense ~]$ passwd
Changing password for user Hank.
Changing password for Hank
(current) UNIX password: [REDACTED]
```

```
[root@sense 26392]# cat /proc/26392/status
Name:      passwd
State:     S (sleeping)
SleepAVG:  88%
Tgid:      26392
Pid:       26392
PPid:      26367
TracerPid: 0
Uid:       500 0 0 0
Gid:       500 500 500 500
FDSize:    256
Groups:    500
VmPeak:    3964 kB
VmSize:    3956 kB
```

Annotations in the image:

- effective**: Points to the first '0' in the 'Uid' field.
- real**: Points to the first '500' in the 'Gid' field.
- saved**: Points to the first '500' in the 'Gid' field.

# Process User ID Model in Modern UNIX Systems

- When a process is created by *fork*
  - it inherits all three users IDs from its parent process
- When a process executes a file by *exec*
  - it keeps its three user IDs unless the set-user-ID bit of the file is set, in which case the effective uid and saved uid are assigned the user ID of the owner of the file
- A process may change the user ids via system calls

# The Need for suid/sgid Bits

- Some operations are not modeled as files and require user `id = 0`
  - halting the system
  - bind/listen on “privileged ports” (TCP/UDP ports below 1024)
  - non-root users need these privileges
- File level access control is not fine-grained enough
- System integrity requires more than controlling who can write, but also how it is written

```

[ bob@foo ]$ cat /etc/passwd
alice:x:1007:1007::/home/alice:/bin/bash
bob:x:1008:1008::/home/bob:/bin/bash
[ bob@foo ]$ cat printid.c

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main() {
    printf(
        "Real      UID = %d\n"
        "Effective UID = %d\n"
        "Real      GID = %d\n"
        "Effective GID = %d\n",
        getuid (),
        geteuid(),
        getgid (),
        getegid()
    );
    return 0;
}

[ bob@foo ]$ gcc -Wall printid.c -o printid
[ bob@foo ]$ chmod ug+s printid
[ bob@foo ]$ su alice
Password:
[ alice@foo ]$ ls -l
-rwsr-sr-x 1 bob bob 6944 2007-11-06 10:22 printid
[ alice@foo ]$ ./printid
Real      UID = 1007
Effective UID = 1008
Real      GID = 1007
Effective GID = 1008
[ alice@foo ]$

```

**United States Patent** [19]**Ritchie**

[11]

**4,135,240**

[45]

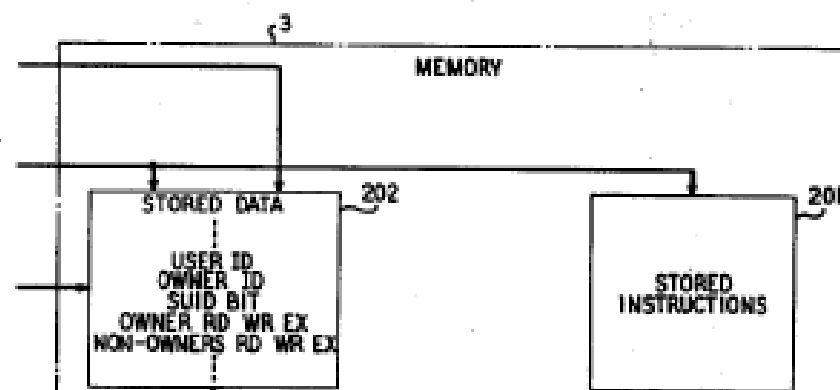
**Jan. 16, 1979**[54] **PROTECTION OF DATA FILE CONTENTS**[75] **Inventor:** Dennis M. Ritchie, Summit, N.J.[73] **Assignee:** Bell Telephone Laboratories,  
Incorporated, Murray Hill, N.J.[21] **Appl. No.:** 377,591[22] **Filed:** Jul. 9, 1973[51] **Int. Cl.<sup>2</sup>** ..... G06F 11/10; G06F 13/00[52] **U.S. Cl.** ..... 364/200[58] **Field of Search** ..... 340/172.5;  
364/200 MS File, 900 MS File[56] **References Cited****U.S. PATENT DOCUMENTS**

Re. 27,239	11/1971	Ulrich .....	340/172.5
Re. 27,251	12/1971	Amdahl et al. ....	340/172.5
3,368,207	2/1968	Beausoleil et al. ....	340/172.5
3,377,624	4/1968	Nelson et al. ....	340/172.5
3,469,239	9/1969	Richmond .....	340/172.5
3,576,544	4/1971	Cordero et al. ....	340/172.5
3,599,159	8/1971	Creech et al. ....	340/172.5
3,631,405	12/1971	Hoff .....	364/200

3,683,418	8/1972	Martin .....	340/172.5
3,735,364	5/1973	Hatta .....	340/172.5
3,742,458	6/1973	Inoue et al. ....	340/172.5
3,761,883	9/1973	Alvarez .....	364/200

*Primary Examiner*—James D. Thomas*Attorney, Agent, or Firm*—Stephen J. Phillips[57] **ABSTRACT**

An improved arrangement for controlling access to data files by computer users. Access permission bits are used in the prior art to separately indicate permissions for the file owner and nonowners to read, write and execute the file contents. An additional access control bit is added to each executable file. When this bit is set to one, the identification of the current user is changed to that of the owner of the executable file. The program in the executable file then has access to all data files owned by the same owner. This change is temporary, the proper identification being restored when the program is terminated.

**4 Claims, 2 Drawing Figures**



# Security Problems of Programs with suid/sgid

- These programs are typically setuid root
- Violates the least privilege principle
  - every program and every user should operate using the least privilege necessary to complete the job
- Why violating least privilege is bad?
- How would an attacker exploit this problem?
- How to solve this problem?

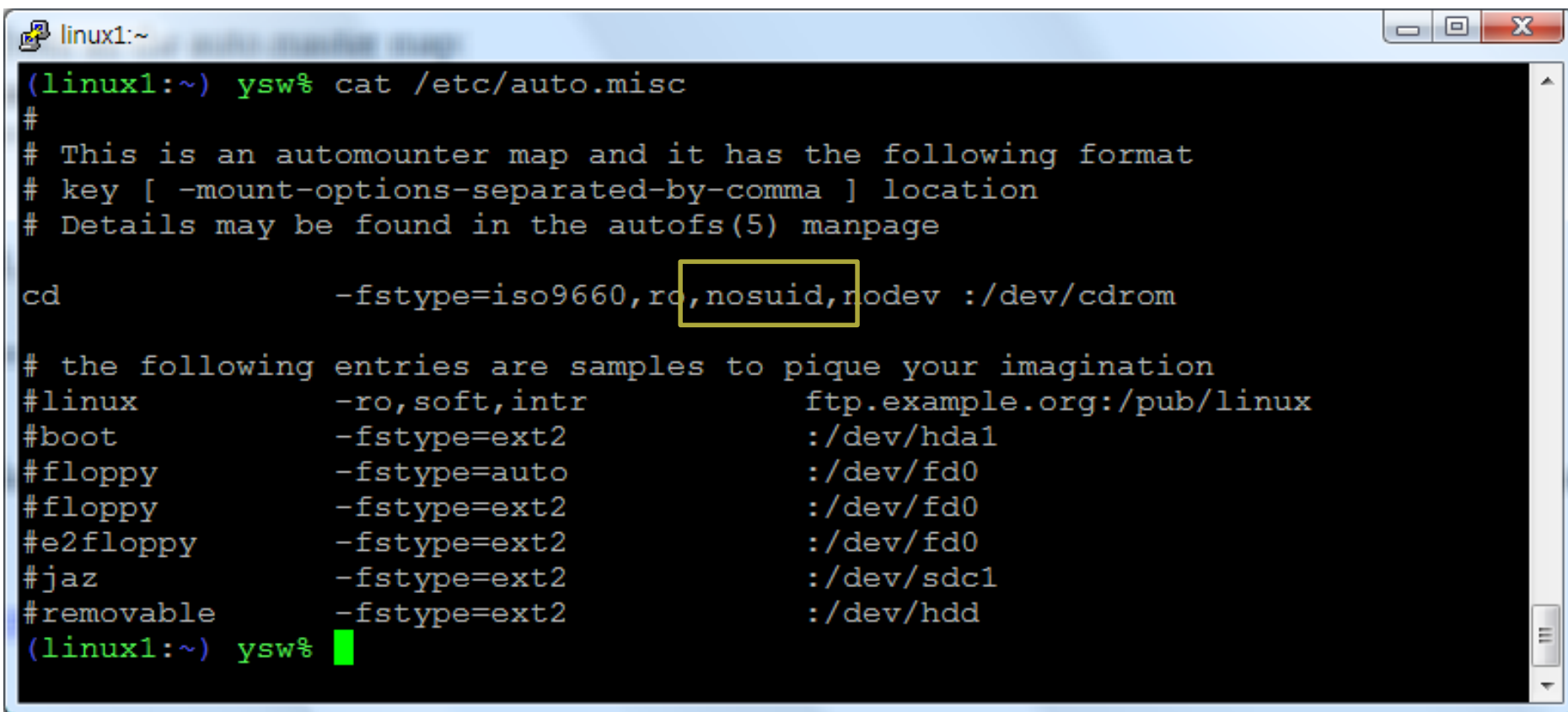
# Security Problems of Programs with suid/sgid

- In general, only /usr partition should allow the suid bit
- USB Flash, external drives, network drives should not allow the suid bit

/dev/sda11	/tmp	ext2	defaults,rw,nosuid,nodev,noexec	1	2
/dev/sda6	/home	ext2	defaults,rw,nosuid,nodev	1	2

(/etc/fstab )

# Security Problems of Programs with suid/sgid

A terminal window titled 'linux1:~' with standard window controls (minimize, maximize, close) in the top right. The terminal shows the command 'cat /etc/auto.misc' being executed. The output is a text file containing comments and a list of automounter entries. One entry, 'cd', is highlighted with a yellow box, showing options '-fstype=iso9660,ro,nosuid,nodev' and a location '/dev/cdrom'. The 'nosuid' option is specifically highlighted with a yellow box. The terminal ends with a prompt '(linux1:~) ysw%' and a green cursor.

```
(linux1:~) ysw% cat /etc/auto.misc
#
# This is an automounter map and it has the following format
# key [ -mount-options-separated-by-comma ] location
# Details may be found in the autofs(5) manpage

cd                -fstype=iso9660,ro,nosuid,nodev :/dev/cdrom

# the following entries are samples to pique your imagination
#linux            -ro,soft,intr          ftp.example.org:/pub/linux
#boot            -fstype=ext2             :/dev/hda1
#floppy           -fstype=auto            :/dev/fd0
#floppy           -fstype=ext2            :/dev/fd0
#e2floppy         -fstype=ext2            :/dev/fd0
#jaz             -fstype=ext2             :/dev/sdc1
#removable        -fstype=ext2            :/dev/hdd
(linux1:~) ysw%
```

# Changing effective user IDs

- A process that executes a set-uid program can drop its privilege; it can
  - drop privilege permanently
    - removes the privileged user id from all three user IDs
  - drop privilege temporarily
    - removes the privileged user ID from its effective uid but stores it in its saved uid, later the process may restore privilege by restoring privileged user ID in its effective uid

```

#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

void PrintID()
{
    uid_t r, e, s;

    getresuid(&r, &e, &s);

    printf("rid=%u eid=%u sid=%u\n", r, e, s);
}

int main()
{
    uid_t old_rid;

    old_rid = getuid();

    printf("Before dropping privilege.\n");
    PrintID();

    seteuid(old_rid);

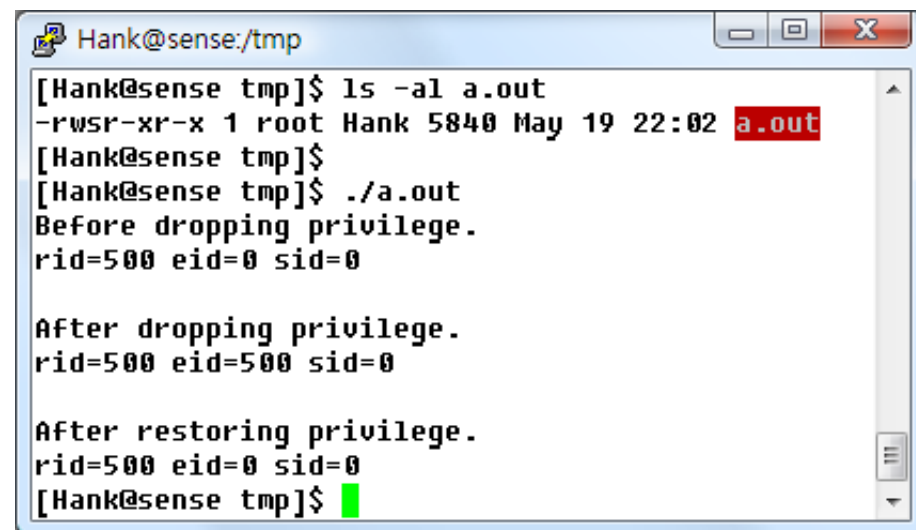
    printf("\nAfter dropping privilege.\n");
    PrintID();

    seteuid(0);

    printf("\nAfter restoring privilege.\n");
    PrintID();

    return 0;
}

```



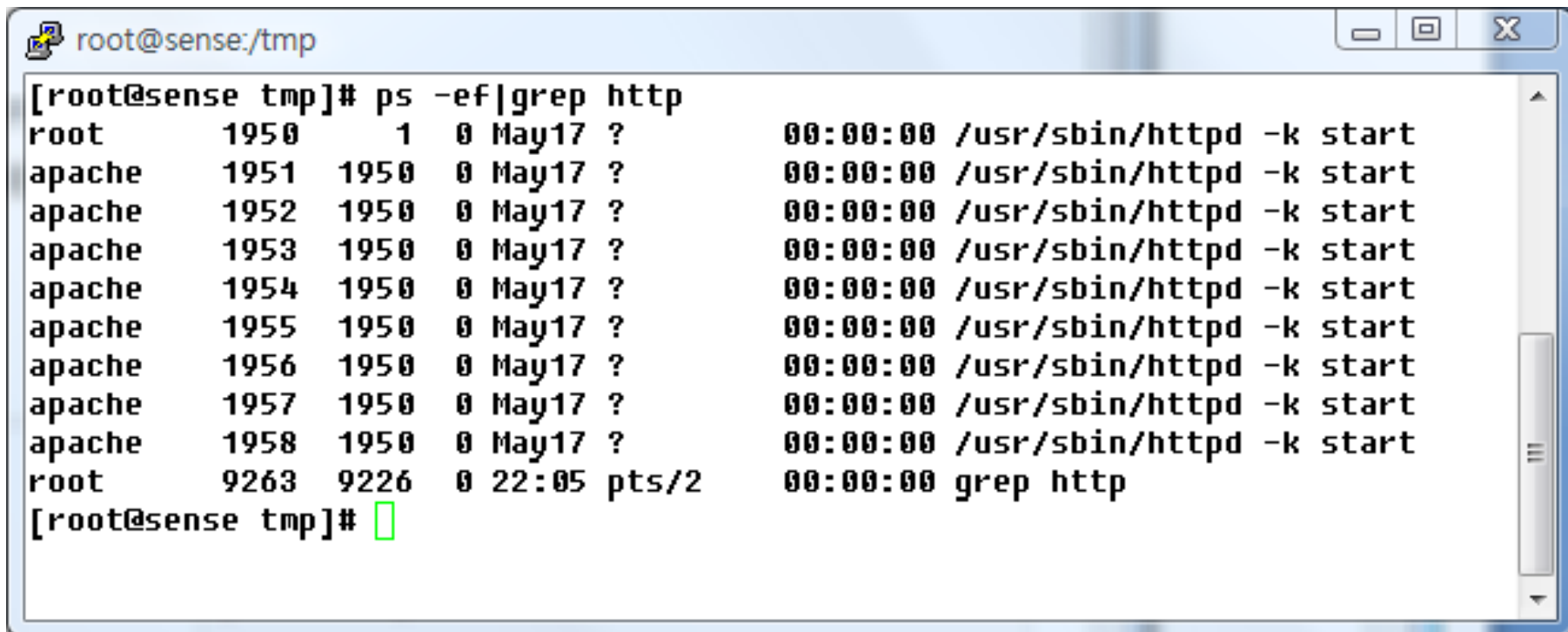
```

Hank@sense:/tmp
[Hank@sense tmp]$ ls -al a.out
-rwsr-xr-x 1 root Hank 5840 May 19 22:02 a.out
[Hank@sense tmp]$
[Hank@sense tmp]$ ./a.out
Before dropping privilege.
rid=500 eid=0 sid=0

After dropping privilege.
rid=500 eid=500 sid=0

After restoring privilege.
rid=500 eid=0 sid=0
[Hank@sense tmp]$

```



```
root@sense:/tmp
[root@sense tmp]# ps -ef|grep http
root      1950      1   0 May17 ?           00:00:00 /usr/sbin/httpd -k start
apache    1951    1950   0 May17 ?           00:00:00 /usr/sbin/httpd -k start
apache    1952    1950   0 May17 ?           00:00:00 /usr/sbin/httpd -k start
apache    1953    1950   0 May17 ?           00:00:00 /usr/sbin/httpd -k start
apache    1954    1950   0 May17 ?           00:00:00 /usr/sbin/httpd -k start
apache    1955    1950   0 May17 ?           00:00:00 /usr/sbin/httpd -k start
apache    1956    1950   0 May17 ?           00:00:00 /usr/sbin/httpd -k start
apache    1957    1950   0 May17 ?           00:00:00 /usr/sbin/httpd -k start
apache    1958    1950   0 May17 ?           00:00:00 /usr/sbin/httpd -k start
root      9263    9226   0 22:05 pts/2       00:00:00 grep http
[root@sense tmp]#
```

# Access Control in Early UNIX

- A process has two user IDs: real uid and effective uid and one system call `setuid`
- The system call `setuid(id)`
  - when `eid` is 0, `setuid` set both the `ruid` and the `eid` to the parameter
  - otherwise, the `setuid` could only set effective uid to real uid
    - Permanently drops privileges
- A process cannot temporarily drop privilege

# System V

- Added saved uid & a new system call
- The system call seteuid
  - if euid is 0, seteuid could set euid to any user ID
  - otherwise, could set euid to ruid or suid
    - Setting to ruid temp. drops privilege
- The system call setuid is also changed
  - if euid is 0, setuid functions as seteuid
  - otherwise, setuid sets all three user IDs to real uid



# BSD

- Uses ruid & euid, change the system call from setuid to setreuid
  - if euid is 0, then the ruid and euid could be set to any user ID
  - otherwise, either the ruid or the euid could be set to value of the other one
    - enables a process to swap ruid & euid

# Modern UNIX

- System V & BSD affect each other, both implemented `setuid`, `seteuid`, `setreuid`, with different semantics
  - some modern UNIX introduced `setresuid`
- Things get messy, complicated, inconsistent, and buggy
  - POSIX standard, Solaris, FreeBSD, Linux

# Suggested Improved API

- Three method calls
  - `drop_priv_temp`
  - `drop_priv_perm`
  - `restore_priv`
- Lessons from this?
- Psychological acceptability principle
  - “human interface should be designed for ease of use”
  - the user’s mental image of his protection goals should match the mechanism