Chapter 5

Operating System Security and Access Control

The official version of this book is available at
`https://www.springer.com/gp/book/9783030834104`

ISBN: 978-3-030-83410-4 (hardcopy), 978-3-030-83411-1 (eBook)

version: 15 Oct 2021

Chapter 5

Operating System Security and Access Control

Mass-produced computers emerged in the 1950s. 1960s time-sharing systems brought security requirements into focus. 1965-1975 was the golden design age for operating system (OS) protection mechanisms, hardware protection features and address translation. While the threat environment was simpler—e.g., computer networks were largely non-existent, and the number of software authors and programs was far smaller—many challenges were the same as those we face today: maintaining separation of processes while selectively allowing sharing of resources, protecting programs from others on the same machine, and restricting access to resources. "Protection" largely meant controlling access to memory locations. This is more powerful than it first appears. Since both data and programs are stored in memory, this controls access to running processes; input/output devices and communications channels are also accessed through memory addresses and files. As files are simply logical units of data in primary memory and secondary storage, access control of memory and files provides a general basis for access control of objects and devices.

Initially, protection meant limiting memory addresses accessible to processes, in conjunction with early virtual memory address translation, and access control lists were developed to enable resource sharing. These remain protection fundamentals. Learning about such protection in operating systems provides a solid basis for understanding computer security. Aside from Unix, we base our discussion in large part on Multics; its segmented virtual addressing, access control, and protection rings heavily influenced later systems. Providing security-related details of all major operating systems is not our goal—rather, considering features of a few specific, real systems allows a coherent coverage highlighting principles and exposing core issues important in any system design. Unix of course has many flavors and cousins including Linux, making it a good choice. Regarding Multics, security influenced it from early design (1964-67) through later commercial availability. It remains among the most carefully engineered commercial systems ever built; an invaluable learning resource and distinguishing feature over both early and modern systems is its rich and detailed technical literature explaining both its motivation and design.

5.1 Memory protection, supervisor mode, and accountability

IN THE BEGINNING. Early computers were large and expensive—and simple compared to later systems. When they were used to run single computer programs one after the other, the delay between runs wasted valuable computer time. This motivated *batch processing*—programs prepared ahead of time were submitted together as a “batched” job run by an operator. This reduced idle CPU time and costs, but inconvenienced users. The *time-sharing systems* of the early 1960s offered an alternative for shared use of a computer, and gave users the impression of running a program on their own machine in real time. While programs appeared to run concurrently, the innovation was to organize them as processes between which the CPU alternated. This is how single-user computers work today, albeit with one user running multitudes of programs concurrently.

ISOLATION. A security issue arises immediately once more than one process runs “concurrently”: resource conflicts. An early concern was computer memory—an isolation mechanism was needed to prevent one process writing into another’s memory, lest benign errors in one program impact another (let’s not mention malicious programs just yet). Even for computers running single programs one at a time, if a user process could access the computer’s full memory range, errors might disrupt OS data or code—a lack of basic protection subjects even a debugging program to disruption by faulty code being debugged. Another early commercial issue was how to allow one program to execute proprietary functionality of a second, without the first having read access to the second.

SUPERVISOR, PRIVILEGED BIT, DESCRIPTOR REGISTER. A typical isolation mechanism began as follows. All memory references went through a hardware *descriptor register* holding a *memory descriptor* consisting of a `(base, bound)` pair of values. These indicated the lowest physical memory address accessible to the active process, and the number of addressable memory words from that point. To control the memory address range visible to a process, only the *supervisor* program, which ran with a *privileged bit* set (supervisor mode), could load the descriptor register. (Earlier machines ran all programs in this mode—it was the only mode.) User programs could cause the privileged bit to be set only via a machine instruction that immediately transferred execution to the supervisor program. The design protected memory descriptors by storing them in memory managed exclusively by the supervisor. This starting basis for process isolation thus consisted of a simple three-component memory protection scheme as follows (Fig. 5.1).

1. *Descriptor register:* constrains the addresses a process can access. The supervisor maintains a descriptor for each process, and loads this register for the active process.
2. *Privileged bit:* must be set in order for the descriptor register to be loaded. Only supervisor code runs with this bit set.
3. *Supervisor:* no other program can alter the privileged bit. A special machine instruction can also set the bit and immediately transfer control to the supervisor.

LIMITATIONS OF MEMORY-RANGE BASED PROTECTION. With this basic memory protection scheme, the supervisor prevents user processes from altering supervisor code or data by reserving memory that user processes cannot access. This provides an *all-or-*

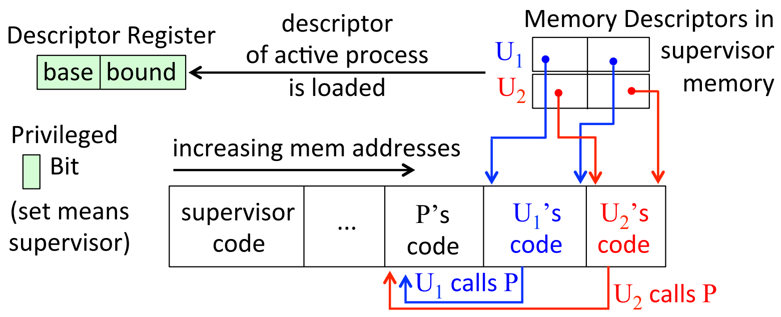

Figure 5.1: Two programs calling the same sub-program P (model circa 1970).

nothing mode in the sense of full access to everything as supervisor, or no cross-process sharing at all. This allows full isolation, but not fine-grained sharing of memory. We next consider why finer-grained control is desirable, while still preventing unauthorized use or alteration of one process's memory by another. Consider a service program P in memory, to be called by user processes U_1 and U_2 (Figure 5.1). Both must access P's code memory. Simple memory-range limits of a descriptor register no longer suffice, e.g., because P needs memory to write temporary results for each of U_1, U_2. The next step forward is more specific *access permissions* allowing separate read, write and execute permissions for a specified memory region. A description of this enhancement follows.

SEGMENT ADDRESSING WITH ACCESS PERMISSIONS. A memory *segment*, supported by *segment addressing hardware*, is a continuous block of words, representing a logical unit of information. (Multics segments evolved from the concept of a *file*, which had already been introduced.) A memory word is then addressed by a pair of values (S,W), the segment number S and word number offset W therein. Thus a level of indirection now separates this early *virtual memory* descriptor from a segment's physical address. The OS maintains a special per-process *descriptor segment* that holds a table of *segment descriptors* defining the physical memory addressable by the process.[1] The addressing scheme controls access—a process can't access a segment that it can't "see" (i.e., reference). A processor *descriptor base register* (DBR) points to the descriptor segment of the active process. S is an index into this table, and each segment descriptor therein details a segment's physical starting address, current size, and an *access control indicator* specifying permission bits for this memory segment, for example (Figure 5.2):

R: *read* (if 1, a non-supervisor process has read access; if 0, only the supervisor does)

W: *write* (if 1, the segment may be written into; usually then X=0)

X: *execute* (if 1, segment is executable; usually then W=0, code not self-modifying)

M: *mode* (if 1, supervisor mode when executing segment; valid only when X=1)

F: *fault* (if 1, all access attempts trap to supervisor; overrides all other bits)

Note that by this design, now the same physical address region can be given (for different processes) different access permissions, through different segment descriptors.

[1]Segment descriptors thus provided early support for the principle of ISOLATED-COMPARTMENTS (P5).

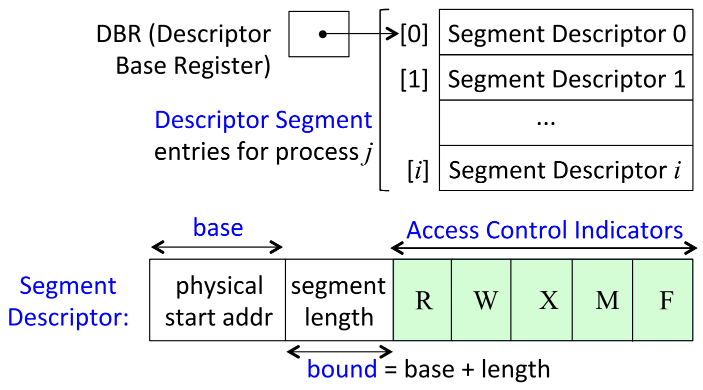

Figure 5.2: Multics-style segment descriptors and a descriptor segment holding them. When process j runs, DBR points to j's descriptor segment, and indexes descriptor i therein as DBR[i]. The index i corresponds to segment S in a memory address pair (S,W).

‡**Exercise** (Memory protection design). In segment addressing above, the access control bits are essentially appended to a base-bound pair to form a descriptor. If the access control indicator bits were instead stored in (a protected part of) the physical storage associated with the target segment, what disadvantage arises?

‡**Permissions on virtual segments.** The segment descriptors above contain physical addresses and access permissions. An improvement associates the access permissions directly with a (virtual) segment identifier, i.e., a label identifying the segment independent of physical memory location. As one motivation, permissions logically relate to virtual, not physical memory. As another, this facilitates combining the resulting (physical) descriptor segment with memory allocation schemes in some designs.

Accountability, userids and principals. Each user account on a system has a unique identifier or `username`, mapped by the OS to a numeric userid (`UID`). To log in (access the account), a user enters the username and a corresponding *password*. The latter is intended to be known only to the authorized user; an expected username-password pair is accepted as evidence of being the legitimate user associated with `username`. This is the basic authentication mechanism, as discussed in Chapter 3. The term *principal* is used to abstract the "entity" responsible for code execution resulting from user (or consequent program) actions. The OS associates a `UID` with each process; this identifies the principal *accountable* for the process. The UID is the primary basis for granting access privileges to resources—the permissions associated with the set of objects the process is authorized to access, or the *domain* in access control terminology.[2] The UID also serves administrative and billing purposes, and aids debugging, audit trails, and forensics. A separate *process identifier* (`PID`) is used for OS-internal purposes such as scheduling.

Roles. A user may function in several roles, e.g., as a regular user and occasionally as an administrator. In this case, by the principle of LEAST-PRIVILEGE (P6), common practice is to assign the user more than one username, and switch usernames (thus UIDs

[2]Section 5.9 defines *subject* (*principal*) more precisely, and the relationship to processes and domains.

internally) when acting in a role requiring the privileges of a different domain; abstractly, distinct UIDs are considered distinct principals. Use of the same username by several users is generally frowned upon as *poor security hygiene*, hindering accountability among other drawbacks. To share resources, a preferred alternative is to use *protection groups* as discussed in Section 5.3. Section 5.7 discusses role-based access control (RBAC).

5.2 The reference monitor, access matrix, and security kernel

Before moving beyond basic protection, we introduce several concepts needed in later sections. The first is the *reference monitor concept* (Figure 5.3), proposed in 1972 as a model for building secure systems for government use in the context of defending against malicious users. The basic notion was stated thus: *all references by any program to any program, data or device are validated against a list of authorized types of reference based on user and/or program function.*

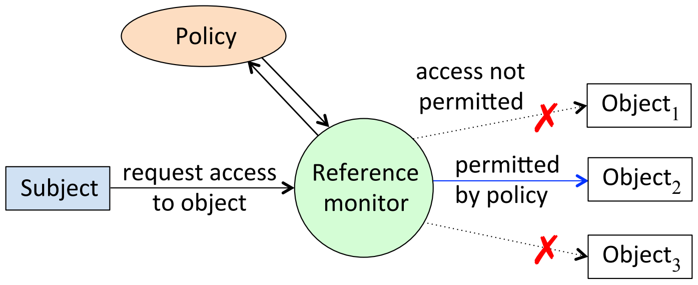

Figure 5.3: Reference monitor model. The policy check consults an access control policy.

Access matrix. The reference monitor is a subject-object model. A *subject* (or *principal*, above) is a system entity that may request access to a system object. An *object* is any item that a subject may request to use or alter—e.g., active processes, memory addresses or segments, code and data (pages in main memory, swapped pages, files in secondary memory), peripheral devices such as terminals and printers (often involving input/output, memory or media), and privileged instructions.

In the model, a system first identifies all subjects and objects. For each object, the types of access (*access attributes*) are determined, each corresponding to an access permission or privilege. Then for each subject-object pair, the system predefines the authorized access permissions of that subject to that object. Examples of types of access are read or write for a data item or memory address, execute for code, wakeup or terminate for a process, search for a directory, and delete for a file. The authorization of privileges across subjects and objects is modeled as an *access matrix* with rows i indexed by subjects, columns j by objects, and entries $A(i, j)$, called *access control entries* (ACEs), specifying access permissions subject i has to object j (Fig. 5.4). The ACE will typically contain a collection of permissions, but we may for ease of discussion refer to the entry as a single permission with value z, and if $A(i, j) = z$ then say that subject i has z-access to object j.

Reference monitor implementation. The reference monitor model is im-

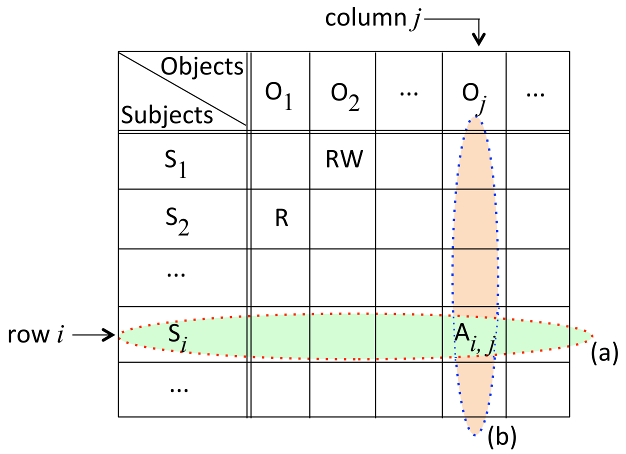

Figure 5.4: Access control matrix. $A(i,j)$ is an ACE specifying the permissions that subject i has on object j. (a) Row i can be used to build a *capabilities list* (C-list) for subject i. (b) Column j can be used to build an *access control list* (ACL) for object j. The access matrix itself is policy-independent; the access control policy is determined by the permission content specified in matrix entries, not the matrix structure.

plemented and enforced by a software-hardware *reference validation mechanism*. It is conceptualized as a single monitor, but in practice would be a collection of monitors each protecting different classes of objects. Every access request made to any system object by any subject is mediated as follows.[3] When subject i attempts z-access to object j, the request information "(i, j, z)" is intercepted by a monitor handling objects of j's class, and the monitor checks whether entry $A(i,j)$ permits z, granting access only if authorized.

REQUIREMENTS. To address known threats, the requirements specified as necessary are that the reference validation mechanism be:

1) tamper-proof;
2) always invoked (not circumventable); and
3) *verifiable*, or more specifically, *"small enough to be subject to analysis and tests, the completeness of which can be assured"*.

These conditions turn out to be difficult to meet in practice. Nonetheless, such a validation mechanism forms the heart of a secure system, and in the context of these requirements is called a *security kernel*. The idea is to centralize, in this nucleus, the minimal control structures and code needed to enforce and verify access control and all other core security-related functions. Even 45 years later, such security kernels are still uncommon—mainstream operating systems remain typically monolithic.[4]

Nonetheless, reference monitor ideas have heavily influenced computer security research and practice, helping popularize the key concepts of access control and access control lists (below).

[3]This is the basis of principle P4 (COMPLETE-MEDIATION) as introduced in Chapter 1.

[4]For example, the monolithic design of the Unix kernel goes against principles P7 (MODULAR-DESIGN) and P8 (SMALL-TRUSTED-BASES).

Reference monitor dependencies. Aside from a properly functioning reference validation mechanism, the reference monitor depends heavily on a number of supporting functions: a trustworthy authentication system (the access matrix assumes legitimate identified subjects), properly operating hardware, physical security of this hardware and system (including storage media and any devices accessing memory), and security of the input-output communication paths between users and the system. For high confidence in the security of a computing system, its entire manufacturing chain, including the production environment of all software and hardware components, and individuals involved, must be trustworthy. The challenge is that complex computer systems require integration of components from countless international suppliers.

Access matrix is a model only. In practice, access control is often implemented by storing permissions within access matrix entries in lists organized either by rows or columns. Mechanisms doing so pre-date the matrix model. Direct implementation of a 2D access matrix is inefficient—the matrix itself is typically large and sparse, while naively storing long lists of entries $(i, j, A(i, j))$ makes searching inefficient.

Capability lists (C-lists). Decomposition by row puts the focus on an individual subject, detailing all access privileges it holds. In Fig. 5.4(a), for a fixed row i^*, taking the non-empty cells (objects j that i^* can access) as a list of tuples $(j, A(i^*, j))$ provides a *capabilities list* (*C-list*) for subject i^*. Each entry specifies i^*'s allowed access permissions on different objects j. Such lists can be held by or associated with individual subjects, referenced as needed. An exercise (page 133) considers implementation issues.

Access control lists. Decomposition by fixed column j^* puts the focus on an individual object; see Fig. 5.4(b). Taking non-empty column cells defines a list of pairs $(i, A(i, j^*))$ ranging down all subjects i permitted access to j^*. Such a list, with entries specifying permitted access modes on j^* by different subjects i, can be constructed, perhaps co-located with j^*, and made available for use on requests to access j^*. This and variations are called *access control lists* (ACLs) for object j^*. An ACE (entry) might conceptually contain fields `(subject; access-type-id = permission-bits)` for a file object, with example values `(adamjones; RWX = 001)` where a 1-bit grants permission. Granting privileges to groups of principals rather than individual subjects may be done using ACEs and Unix-like protection groups (Section 5.3).

‡**Exercise** (Access control of ACLs). (a) Is an ACE itself subject to access control? (b) In systems where a file creator controls the file's ACE, how can a user process alter the ACE if the ACE is stored in supervisor memory? (c) A disadvantage of creator-owner schemes is that if the object owner is unavailable to alter permissions, there is no elegant way to reassign access control. In one *hierarchical access control* alternative, the principal creating an object designates a distinct primary access control principal, and a secondary access control principal subordinate to it in an access control hierarchy. From this sketch, outline a full working scheme. (Hint: [29, Fig.13].)

Capabilities vs. ID-based protection. Protection mechanisms may be *ticket-oriented* or *ID-based*. The first model—implemented using *capabilities*—is that of a ticket or access token (*bearer token*) allowing entry to an event, regardless of the ticket holder's identity, provided that the ticket is recognized as authentic. In the second model, a guard at the door does an identity check, e.g., using photo ID cards in the physical world, and any entity whose identity is verified and on an authorized list is allowed access. Capabilities (tickets) are held by subjects; authorization lists (based on identity) are held by an object's guard. Tickets must be unforgeable; identities must be unspoofable.

‡**Exercise** (Implementing capabilities). To prevent unauthorized copying or tampering of *capabilities* by users, capability systems may be implemented using different options. a) Maintain data structures for capabilities in OS (kernel) memory. b) Maintain data structures for capabilities in user memory, but with tamper protection by a message authentication code (MAC) or equivalent. c) Rely on hardware support (one scheme uses kernel-controlled *tag bits* designating memory words that hold capabilities). Explain further details for each of these mechanism approaches. (Hint: [35]; see also [38].)

‡**Exercise** (Access control and USB drives). When a USB flash drive is inserted into a personal computer, which system accounts or processes are given access to the content on this storage device? When files are copied from the USB drive into the filesystem of an account on the host, what file permissions result on the copied files (assume a Unix-type system)? Discuss possible system choices and their implications.

Basis for audit trails. The basic reference monitor idea of mediating every access provides a natural basis from which to build fine-grained audit trails. Audit logs support not only debugging and accountability, but intruder detection and forensic investigations. Whether or not audit records must be tamper-proof depends on intended use.

‡**Exercise** (Access control through encryption and key release). Access control to documents can be implemented through web servers and encryption. Give a technical overview of one such architecture, including how it supports an audit trail indicating access times and subjects who access documents. (Hint: [11].)

5.3 Object permissions and file-based access control

The *access control indicator* bits in Section 5.1 were a good start, helping us understand basic issues.[5] Early protection based on memory segments gave way to setting permissions on abstract objects, and use of access control lists (Section 5.2). For object-based access control in a subject-object permission framework, after specifying subjects and objects, the task is to identify the types of access operations (modes) for objects and frame these as permissions for consideration.

File-based access control. A common approach to learn about object-level access permissions is to consider logical files—for pedagogical reasons and concrete implementation examples. The next few sections discuss file-based access control as found in Unix systems. Beyond a file's data contents, filesystems maintain per-file meta-data specifying access permissions. In Multics and Unix, an early design principle was to treat everything as a file, and design a corresponding filesystem; this simplifies input-output operations across a multitude of peripheral devices. For example, if printing a file is done by writing a stream of bytes to an address identified with a printing device, then access

[5]Filesystem management of access control was already mentioned in 1967 [14], related to early Multics.

permissions to the printer are "file permissions". Thus the study of file permissions generalizes to access control on resources. This explains why file permissions are a main focus when access control is taught.

ACL ALTERNATIVES. The simple permission mechanisms in early systems provided basic functionality. Operating systems commonly now support ACLs (Section 5.2) for system objects including files. ACLs are powerful and offer fine-grained precision—but also have disadvantages. ACLs can be as long as the list of system principals, consuming memory and search time; ACLs may need frequent updates; listing all principals requiring access to a file can be tedious. A less expressive alternative, the `ugo` architecture, became popular long ago, and remains widely used; we discuss it after background on file ownership. Section 5.7 discusses a further alternative: role-based access control.

FILE OWNER AND GROUP. Unix-based systems assign to each file an *owner* and a *protection group*, respectively identified by two numeric values: a userid (UID) and a *groupid* (GID). Initial values are set on file creation—in the simplest case, using the UID and GID of the creating process. Where are these from? The login username for each account indexes an entry in world-readable Unix file */etc/passwd*, of the form:

```
username:*:UID:GID:fullname_or_info:home_dir:default_shell
```

Here GID identifies the *primary group* for this username. The `*` is where the password hash, salt, and related information was historically stored; it is now typically in */etc/shadow*, readable only by `root`. Group memberships are defined in a distinct file, */etc/group*, each line of which lists a groupname (analogous to a username), a GID, and the usernames of all group members.

SUPERUSER VS. ROOT. Other than for login, the system uses UID for access control, not username. *Superuser* means a process running with UID=0; such a process is granted access to all file resources, independent of protection settings. The username conventionally associated with UID=0 is "root", but technically the string "root" could be assigned to other UIDs. It is the UID value of 0, not the string name, that determines permissions. Within this book we will assume `root` is synonymous with superuser (UID=0).

USER-GROUP-OTHERS MODEL. We can now explain the base architecture for Unix file permissions.[6] The `ugo` *permission model* assigns privileges based on three categories of principals: `(user, group, others)`. The `user` category refers to the principal that is the file owner. The `group` category enables sharing of resources among small to medium-sized sets of users (say, project groups), with relatively simple permissions management. The third category, `others`, is the universal or world group for "everyone else". It defines permissions for all users not addressed by the first two categories—as a means to grant non-empty file permissions to users who are neither the file owner nor in the file's `group`. This provides a compact and efficient way to handle an object for which many (but not all) users should be given the same privileges. This `ugo` model allows fixed-size filesystem meta-data entries, and saves storage and processing time. Whereas ACLs may involve arbitrary-length lists, here permission checking involves bit-operations on sets of just three categories of principals; the downside is a significant loss in expressiveness.

[6]This may be viewed as supporting principle P4 (COMPLETE-MEDIATION).

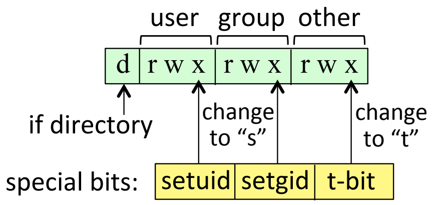

Figure 5.5: Symbolic display of file permissions. `user` is file owner; t-bit is the *sticky bit*. To compactly display a special bit set to 1, `x` becomes `s` or `t` (or resp., `-` becomes `S` or `T`).

Meta-data and file permissions. The above user-group-others mechanism is supported by a per-file filesystem data structure, which also holds other "accounting details" related to a file, such as the address of the file contents. A commonly used such data structure contains the following protection-related fields:[7]

1) `user`: indicating the userid (UID) of the file owner.
2) `group`: indicating the groupid (GID) of the file.
3) 9 bits: 3 protection bits for each of `(user, group, others)` as shown in Fig. 5.5. For regular files, their meaning is fairly straightforward, as follows.
   - R (read): the file content may be read.
   - W (write): an existing file's content may be modified.
   - X (execute): a binary file may be run. To run a shell script requires R to read the file plus X. For a non-executable file, X is not useful (execution fails).
4) 3 bits: special protection bits `setuid`, `setgid`, `t-bit` (Sections 5.4 and 5.5).

Use of protection bits. When a user process requests access to a file, the system checks whether the process has the requested access privilege, based on the permissions in this data structure. The checks are made in sequence: user, group, others. The first qualifying category determines privileges. For a process that seeks R access and is the file owner, if the `user` category does not grant R, the request fails even if `others` grants R.

Permission display notation. A common visual display format for file permissions is a 10-character string, such as `-rwxr-xr--` (Table 5.1). The first character conveys file type—a leading dash indicates a non-directory file. The next nine characters, in groups of three, convey permissions for the `ugo` categories in order. A substring `rwx` corresponds to binary `111` indicating read, write and execute, while a dash "`-`" conveys a `0`-bit denoting that the corresponding permission is absent. Additional permissions can be overlaid onto these ten characters, as shown in Figure 5.5.

Protection bit initial values (non-directory files). A file's 9 RWX bits are set at file creation using a 12-bit mode parameter (with the 3 special bits) provided to the syscall used to create the file. The requested RWX bits are post-modified by the creator's 9-bit file creation "unset" mask (`umask`), typically defined in a user startup file; a child process inherits its parent's umask. The mask's positional 1-bits specify permissions to turn off or remove (if present). For example for regular files—using octal format, with

[7]This is based on the Unix `inode` (index-node), to illustrate concepts concretely in this section. Other `inode` fields unrelated to permissions indicate: file size, last-modified time, number (*link count*) of directory entries that link to the file, and fields to signal whether the file is a directory or special Unix I/O device file.

Binary (12 bits)	Octal	Symbolic	Meaning
`000 100 000 000`	`0400`	`- r-- --- ---`	`user` (owner) has R
`000 010 000 000`	`0200`	`- -w- --- ---`	`user` (owner) has W
`000 001 000 000`	`0100`	`- --x --- ---`	`user` (owner) has X
`000 000 110 000`	`0060`	`- --- rw- ---`	`group` has R, W
`000 000 101 000`	`0050`	`- --- r-x ---`	`group` has R, X
`000 000 011 000`	`0030`	`d --- -wx ---`	`group` has W, X; file is a directory file
`000 000 000 111`	`0007`	`- --- --- rwx`	`other` has R, W, X
`000 110 100 100`	`0644`	`- rw- r-- r--`	`user` has R, W; `group` and `other` have R

Table 5.1: Symbolic notation to display Unix file permissions R/read, W/write, X/execute. Combinations shown are to explain notation, rather than useful combinations. In symbolic notation, a leftmost "`d`" indicates a directory (rather than a permission). The three *special bits* (leftmost) are displayed symbolically by modifying an "`x`" letter as per Fig. 5.5.

one octal digit for each 3-bit RWX group (Table 5.1)—a common permissions default `666` (RW for all categories), and common mask of `022` (removing W from group and others), yield a combined initial permission string `644` (RW for user, R only for group and others).

Example *(group permissions).* Suppose a group identifier `accounting` is set up to include `userA`, `userB` and `userC`, and a group `executives` to include `userC`, `userD` and `userE`. Then `userC` will have, e.g., RX access to any file-based resource if the file's `group` is `accounting` or `executives`, and the file's `group` permissions are also RX.

Exercise (setting/modifying file permissions). The initial value of a Unix mask can be modified where set in a user startup file, or later by the **umask** command. (a) Experiment to discover default file permissions on your system by creating a new file with the command **touch**, and examining its permissions with `ls -l`. Change your mask setting (restore it afterwards!) using **umask** and create a few more files to see the effect. (b) The command **chmod** allows the file owner to change a file's 9-bit protection mode. Summarize its functionality for a specific flavor of Unix or Linux.

Exercise (modifying file owner and group). Unix commands for modifying file attributes include **chown** (change owner of file) and **chgrp** (change a file's group). Some systems allow file ownership changes only by superuser; others allow a file owner to **chown** their owned files to any other user. Some systems allow a file owner to change the file's group to any group that the file owner belongs to; others allow any group whatsoever. Summarize the functionality and syntax of these commands for a chosen Unix flavor.

Exercise (access control in swapped memory). *Paging* is common in computer systems, with data in main memory temporarily stored to secondary memory ("swapped to disk"). What protection mechanisms apply to swapped memory? Discuss.

File permissions augmented by ACLs. The `ugo` permission architecture above is often augmented by ACLs (Section 5.2). On access requests, the OS then checks whether the associated `UID` is in an ACL entry with appropriate permissions.

Exercise (file ACL commands). For a chosen Unix-type system (specify the OS version), summarize the design of file ACL protection. In particular, explain what information is provided by the command **getfacl**, and the syntax for the **setfacl** command.

5.4 Setuid bit and effective userid (eUID)

SETUID BIT. The setuid bit is one of three "special" permission bits (Fig. 5.5). A Unix file owner can turn on this bit for any binary executable file owned, say *file1*. Later when a process with X permission thereon runs *file1*, the OS will temporarily set—while executing *file1*—that process' (effective) userid to be that of *file1*'s owner. This allows *file1* to access resources that the calling process might not itself have sufficient privilege to access. The bargain made is that the calling process now has more access than it otherwise would, but only under the constraint of a carefully designed (hopefully trustworthy) fixed-functionality program. A major use is for programs that access security-critical system resources. This privilege model makes such programs of special interest to attackers, especially when the file owner is root.[8] A setuid program owned by a regular user is also of use, e.g., to allow others controlled access to that user's files; analogous setgid programs (below) allow shared file access within groups.

REAL USERID AND EFFECTIVE USERID. To support the setuid bit and related functions, the OS tracks three process-related userid values: a real userid (rUID) denoting the process owner, an effective userid (eUID), and a saved userid (sUID) to facilitate switching between the previous two, e.g., to selectively drop privileges (changing eUID to rUID) so that higher privileges are active only while executing code segments that require them—in line with principle P6 (LEAST-PRIVILEGE). The eUID determines privileges on resource access requests, and is used to set the file owner on created files. Supporting system calls getuid() and geteuid() respectively return rUID and eUID. Privileged functions setuid(), seteuid() and setreuid() allow setting rUID, eUID and/or sUID; details are beyond our scope. As a use example, the command **su** (substitute user identity) executes a new shell with rUID, eUID, sUID set to the userid of the specified user.

‡**Exercise** (sudo command). Look up and explain the design and use of the Unix command **sudo** (substitute user do), including its effect on rUID, eUID, sUID.

SETGID PERMISSION BIT. For non-directory files, setgid is analogous to setuid, but conveys privileges for a file's protection group, using corresponding system values rGID, eGID, sGID and supporting system calls getgid(), getegid(), setgid(). Section 5.5 explains functionality of the setgid bit for directory files.

Example *(setuid and passwd).* Unix users initiate password changes with the **passwd** command. On many systems, */usr/bin/passwd* is root-owned and has the setuid bit set.[9] This enables write access to the system password file—but the program checks rUID, to enforce that only the password entry for the UID of the calling process can be changed.

INHERITING USERIDS. Process creation in Unix occurs by the syscall fork(). This replicates the original process (*parent*), creating a *child*. Both run the same code. The child inherits the parent userids (rUID, eUID, sUID), but gets a new process ID (PID). If the child process was forked in order to run a separate executable, it recognizes itself as a child by seeing a fork() return code of 0 and cedes control, via one of the exec()-family

[8]Concerns arise from the design principles violated (Section 5.2) by Unix's monolithic kernel design.

[9]On other systems this executable itself is not "setuid"; Mac OS X uses an alternate means.

library calls,[10] to the specified executable, which continues with the child's PID and also inherits its userids (rUID, eUID, sUID) except when the executable is setuid.

Example *(userids and login).* The Unix command **login** results in a process running the root-owned setuid program */bin/login*, which prompts for a username-password, does password verification, sets its process UID and GID to the values specified in the verified user's password file entry, and after various other initializations yields control by executing the user's declared shell. The shell inherits the UID and GID of this parent process.

DISPLAYING SETUID AND SETGID BITS. If a file's setuid or setgid bit is set, this is displayed in 10-character notation by changing the user or group execute letter as follows (Figure 5.5): an x changes to s, or a "-" changes to S. The latter conveys status, but is not useful—setuid, setgid have no effect without execute permission. So -rwsr-xr-x indicates a file executable by all, with setuid bit set; and -rwSr--r-- indicates a file is setuid, but not executable (which is not useful; the unusual capital S signals this).

‡**Exercise** (setuid). Explain how the setuid functionality is of use for user access to a printer daemon, and what general risks setuid creates. (Chapter 6 discusses privilege escalation related to setuid programs in greater detail.)

5.5 Directory permissions and inode-based example

Here we discuss how the permission bits on directory files have different semantics than for regular files. Our discussion is informed by an explanation of how Unix filesystems are implemented, with a detailed example.

UNIX DIRECTORY STRUCTURE. A Unix-style filesystem is a rooted tree with top named "/". Regular files are leaf nodes; each interior node is a *directory file* (dirfile). Both file types are implemented with meta-data stored in an inode (Section 5.3) plus a datablock for content. Meta-data relevant to our discussion are: permissions data, a flag distinguishing dirfiles, and a pointer (datalink) to the datablock. A dirfile's datablock contains filesystem data for the entries (hierarchical children) of that directory node, structured as a list of entries dir-entry = (d-name, d-inode). A string value populating d-name names a (regular or directory) file; the d-inode value identifies that file's inode. The first two dir-entries are always for the directory node itself (denoted ".") and its parent directory or parent-dir (denoted ".." by convention). See Fig. 5.6 on page 140.

DIRECTORY PERMISSIONS. Section 5.3 introduced the 12 permission bits for regular (non-directory) files. For directory files, these have very different meanings, perhaps better reflected by calling them LAT bits (List, Alter, Traverse). We now summarize their meaning, for a user with the specified permissions on a directory (e.g., having R permission for the first category user, group or other that applies). Note: as for regular files, the superuser may access directory files regardless of permission bits.

R: the user may *list* the directory content, i.e., view the filenames that are d-name fields of dir-entry items (Fig. 5.6). R alone on a directory gives no access to the content

[10]Chapter 6 provides further background on fork() and exec().

of files therein (they have their own permissions), nor their meta-data.

W: the user may *alter* (edit) directory content—provided X permission is also held. W (with X) allows renaming or deleting filename entries, and creating new entries (by creating a new file, or linking to an existing file as explained in Section 5.6); the system will modify, remove, or newly add a corresponding `dir-entry`. Thus removing a file reference (entry) from a directory requires not W permission on the target file, but W (with X) on the referencing directory.

X: the user may *traverse* and "search" the directory. This includes setting it as the working directory for filesystem references; *path-access*[11] to the directory's listed files; and access to their inode meta-data, e.g., by commands like **find**. Lack of X on a directory denies path-access to its files, independent of permissions on those files themselves; their filenames (as directory content) remain listable if R is held.

`setuid`: this bit typically has no meaning for directory files in Unix and Linux.

`setgid`: if set, the `group` value initially assigned to a newly created (non-directory or directory) file is the GID of its directory (rather than the GID of the creating process); a newly created sub-directory in addition inherits the directory's `setgid` bit. The aim is to make group sharing of files easier, by setting a directory's setgid bit.

`t-bit`: (*text* or *sticky* bit) this bit set on a directory prevents deletion or renaming of files therein owned by other users. The directory owner and `root` can still delete files. For non-directory files, the `t-bit` is now little used (its original use is obsolete).

STICKY BIT. A primary use of the `t-bit` is on world-writable directories, e.g., */tmp*. An attack could otherwise remove and replace a file with a malicious one of the same filename. When set, a `t` replaces `x` in position 10 of symbolic display strings (Fig. 5.5).

Example *(Directory permissions).* In Fig. 5.6, *curry* and *durant* are entries in *Warriors*. Path-access to *Warriors*, including X on the inode it references (Warriors_inode), is needed to make this the current directory (via `cd`) or access any files below it. R on Warriors_inode allows visibility of filenames *curry* and *durant* (e.g., via `ls`). X on Warriors_inode allows access to the meta-data of *curry* and *durant* (e.g., via `ls -l` or `find`), but to read their content requires R on these target file inodes (plus X on Warriors_inode, in order to get to them). In summary: access to a file's name (which is a directory entry), properties (inode meta-data including permissions), and content are three distinct items. Access to a (dir or non-dir) file's meta-data requires path-access (X) on the inode holding the meta-data, and is distinct from RWX permission on the file content referenced by the inode.

WORLD-WRITABLE FILES. Some files are writable by all users (*world-writable*). This is indicated by `w` in the second-last character for a display string such as `-rwxrwxrwx`. The leading dash indicates a non-directory (regular) file. Some files should not be world-writable, e.g., a *startup file* (a script file invoked at startup of a login program, shell, editor, mail or other executable), lest an attack modify it to invoke a malicious program

[11]By Unix *path-based permissions*, to read a file's content requires X on all directories on a path to it plus R on the file; path-access does not require R on directories along the path if the filename is already known.

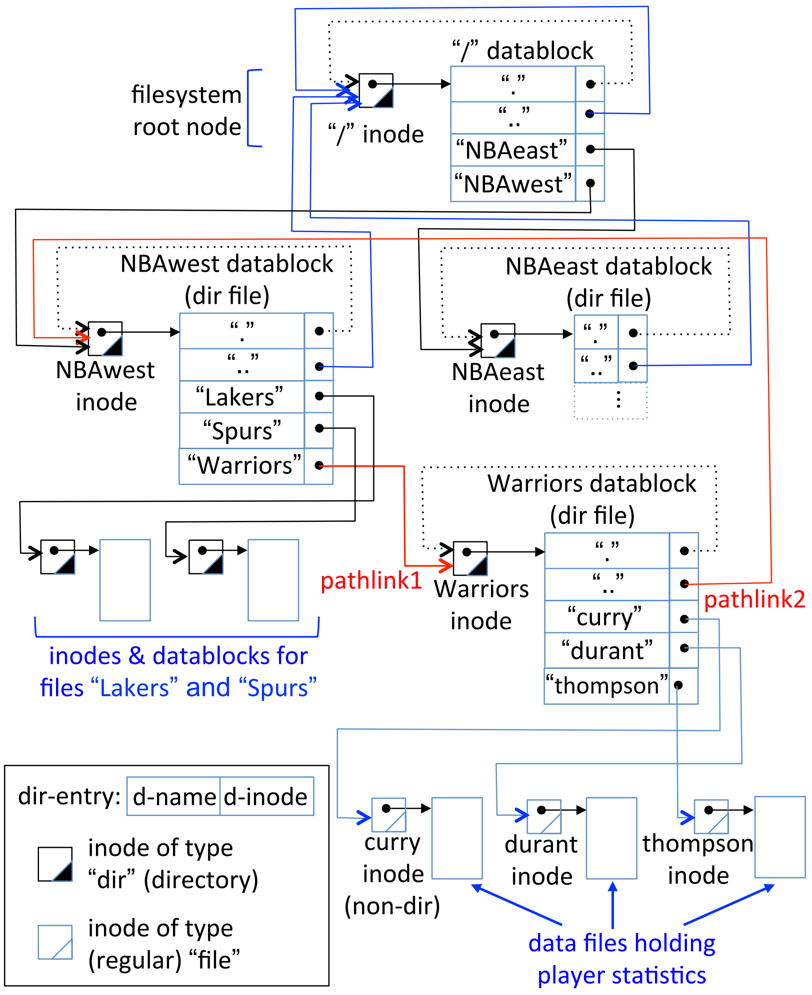
filesystem root node
"/" datablock
"/" inode
"."
".."
"NBAeast"
"NBAwest"
NBAwest datablock (dir file)
NBAwest inode
"Lakers"
"Spurs"
"Warriors"
NBAeast datablock (dir file)
NBAeast inode
Warriors datablock (dir file)
Warriors inode
pathlink1
pathlink2
"curry"
"durant"
"thompson"
inodes & datablocks for files "Lakers" and "Spurs"
dir-entry: d-name d-inode
inode of type "dir" (directory)
inode of type (regular) "file"
curry inode (non-dir)
durant inode
thompson inode
data files holding player statistics

Figure 5.6: Directory structure and example (Unix filesystem with inodes). See inline discussion regarding the roles of `pathlink1` and `pathlink2`. Compare to Fig. 5.7.

on each startup. A world-writable root-owned setuid executable is spectacularly bad—any process could replace the contents with a malicious file, which would run with root privileges regardless of who invoked it. (Chapter 6 has a further example on this topic.)

‡**Exercise** (finding world-writable files). The Unix **find** command can search for files with specific properties, including permissions. Explore your own system for world-writable files using: `find` */users/yourhome* `-perm -2 -print` (replace */users/yourhome* with a directory to explore; start at one with a relatively small subtree, as the output is often extensive). The search will be denied access to directories that you lack X permission for; the command will continue with the next directory in a recursive tree search.

‡**Example** *(inode details).* The following, and Figure 5.6, explain the internal actions that occur in creating a new directory node, and provide a detailed example of the filesystem (directory) structure summarized above. This may help in understanding how

directory-node RWX permissions work. Suppose that in an existing directory */NBAwest*, we wish to create a new sub-directory */NBAwest/Warriors*. Peer sub-directories *Lakers* and *Spurs* already exist. While in current directory */NBAwest*, a user types the shell command: `mkdir Warriors`. The resulting OS actions begin with a system call to mknod() with parameters indicating filepath */NBAwest/Warriors* and that this should be a new directory. The following OS actions then also occur.

1) Creation of a new `inode` instance for *Warriors*, flagged as a directory, initially with null datalink. Let `pathlink1` be a pointer to this new inode.
2) Creation of a new datablock with two directory entries, `(".", NULL)` and `("..", NULL)`. The null datalink of the inode at `pathlink1` is set to point to this datablock.
3) So that the new directory node for *Warriors* can be reached from its parent-dir, the entry `("Warriors", pathlink1)` is added to the list of dir-entries in the datablock of that parent-dir. This datablock will already have entries for `"."` and `".."`, plus peers `"Lakers"` and `"Spurs"` as noted above. The system can obtain a pointer (call it `pathlink2`, for use also below) to the parent-dir inode from available parameters.
4) The system makes two system calls to link() to fix the two null dir-entry pointers above. The first results in the datablock pointer for `"."` in the *Warriors* datablock dir-entry being set to `pathlink1`, i.e., pointing to its own inode; the second results in the datablock pointer for `".."` in the *Warriors* datablock dir-entry being set to `pathlink2`, i.e., pointing to its parent inode, the inode for *NBAwest*. So the datablock entries are now `(".", pathlink1)` and `("..", pathlink2)`.

The user now creates (regular) files named *curry*, *durant*, *thompson*, etc., for storing player statistics. This ends our example—wasn't reading through all that great fun! If you found it hard to follow, read it once more—but start with a blank sheet in front of you, and draw out what happens at each step. (Who knew computer security could be so enjoyable!)

UNEXPECTED DETAIL. The string `"Warriors"` appeared nowhere in the two data structures representing the logical file *Warriors*. It appears only in the parent's dir-entry for this file. Thus one file can be given different names by different parents (Section 5.6).

PROTECTION BIT INITIAL VALUES (DIRECTORY FILES). As for regular files (above), when a new Unix directory is created, its 9 RWX bits are assigned. A common system default for directories is `777`, and for the common default mask `022`, the combined default permission for a directory is then `755` (RWX for user, RX for group and others).

Example *(Directory listings).* Typing `ls -l` to a Unix shell lists the contents of a directory (`-l` for long format, including permissions); use `ls -ld` for meta-data on the directory node itself. As before, the output is a sequence of lines each beginning with a 10-character sequence such as `drwxrwxr-x`, each line giving information about a file listed. A leading `d` signals a directory file. The next nine characters, in groups of three, reflect permissions for the categories `user, group, others`; dash indicates no privilege. The string `drwxrwsr-x` indicates a `setgid` bit set for a directory file.

Exercise (viewing directory properties). On a Unix-like system, explore the properties and protection bits of various directories using the **ls** (list) command and various options

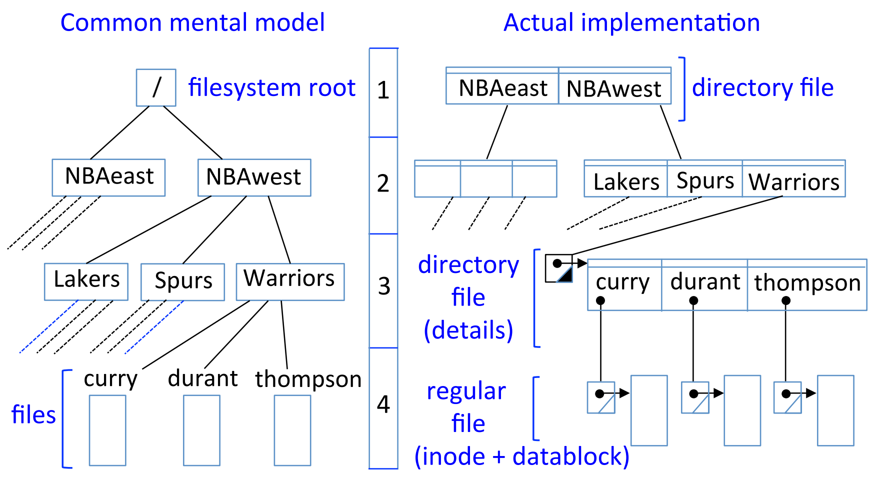

Figure 5.7: Common mental model of a Unix directory structure. As typically implemented, a file's inode and datablock themselves contain no filename. Instead, they are data structures referenced by a directory entry, which specifies the filename from one specific directory-file's view; this allows the same structure to be referenced by different names from distinct (multiple) directory entries. The poor fit of mental model to implementation may lead to misunderstanding directory permissions. Compare to Fig. 5.6.

like `ls -l`. (Note: on a Mac system, to get a command interpreter shell to the underlying Unix system, run the Terminal utility, often found in *Applications/Utilities*.)

Exercise (access control outside of filesystem). Suppose a copy of a filesystem's data (as in Figure 5.6) is backed up on secondary storage or copied to a new machine. You build customized software tools to explore this data. Are your tools constrained by the permission bits in the relevant inode structures? Explain.

‡**Exercise** (chroot jails). A *chroot jail* provides modest isolation protection, supporting principle P5 (ISOLATED-COMPARTMENTS) through restricted filesystem visibility. It does so using the Unix system call chroot(), whose argument becomes the filesystem root ("/") of the calling process. Files outside this subsystem are inaccessible to the process. This restricts resources and limits damage in the case of compromise. Common uses include for network daemons and guest users. a) Summarize the limitations of chroot jails. b) Describe why the newer jail() is preferred, and how it works. (Hint: [17].)

5.6 Symbolic links, hard links and deleting files

A number of security issues arise related to file links, motivating discussion of them here. In Unix, the same non-directory file can appear in multiple directories, optionally with different names. This is done by *linking* using the **ln** command. A link can be either a *symlink* (symbolic/soft link or *indirect alias*) or a *hard link* (*direct alias*).

Example *(Hard link).* Consider a file with pathname *existing*. Then the command `ln existing new1` results in a `dir-entry` specifying the string *new1* as the name of a file object whose directory entry references the inode for *existing*. Since a file's name is not part of the file itself, distinct directory entries (here, now two) may name the file differently. Also, this same command syntax may allow a directory-file to be hardlinked (i.e., *existing* may be a directory), although for technical reasons, hardlinking a directory is usually discouraged or disallowed (due to issues related to directory loops).

Example *(Symlink).* For a symlink the `-s` option is used: `ln -s existing new2` results in a `dir-entry` for an item assigned the name *new2* but in this case it references a new inode, of file type `symlink`, whose datablock provides a symbolic name representing the object *existing*, e.g., its ASCII pathname string. When *new2* is referenced, the filesys-

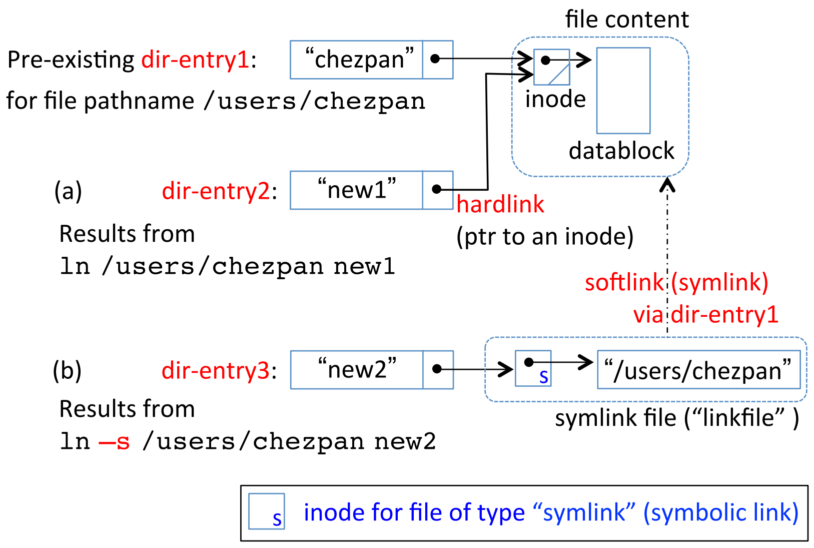

Figure 5.8: Comparison of (a) hardlink, and (b) symbolic link. A symbolic link can be thought of as a file whose datablock itself points to another file.

	... what happens to:	
If ↓ then ...	Hardlink file	Symlink file
target file deleted ("rm dir-entry1" in Figure 5.8)	dir-entry1 is removed, but file content does not disappear since its link count is above 0	softlink will fail (dir-entry3 of symlink remains but is stale, i.e., can't be resolved)
target file renamed or moved to a new pathname	hardlink remains intact, as inode has not changed (nor moved); only the dir-entry1 changes	softlink will fail as target pathname can't be resolved
symlink file's dir-entry deleted ("rm new2")	target file referred to by symbolic link is unaffected	linkfile inode, its data and dir-entry3 disappear, but target file is unchanged

Figure 5.9: Results of deleting, renaming/moving, and removing linked files. "Deleting" a file removes its dir-entry, but does not always result in the file content objects being deleted (see table). Here the "target file" is `/users/chezpan` in Figure 5.8.

tem uses this symbolic name to find the inode for *existing*. If the file is no longer reachable by pathname *existing* (e.g., its path or that directory entry itself is changed due to renaming, removed because the file is moved, or deleted from that directory), the symbolic link will fail, while a hardlink still works. Examining Figure 5.8 may help clarify this.

DELETING LINKS AND FILES. "Deleting" a file removes the filename from a directory's list, but the file itself (inode and datablock) is removed from the filesystem only if this was the last directory entry referencing the file's inode.[12] Figures 5.8 and 5.9 consider the impact of deletion with different types of links. Deleting a `symlink` file (e.g., *new2* directory entry in Fig. 5.8) does not delete the referred-to "file content". Deleting a hardlinked file by specifying a particular `dir-entry` eliminates that directory entry, but the file content (including inode) is deleted only when its *link count* (the number of hardlinks referencing the inode) drops to zero. While at first confusing, this follows directly from the filesystem design: an inode itself does not "live" in any directory but rather exists independently; directory entries simply organize inodes into a structure.

5.7 Role-based (RBAC) and mandatory access control

Here we give a brief overview of role-based access control, discuss how the discretionary access control approach commonly found in commodity operating systems differs from mandatory access control, and mention SELinux as an example of the latter.

MANDATORY AND DISCRETIONARY ACCESS CONTROL. Access control policy rules are enforced by the operating system. Multics and Unix-style file permissions are examples of *discretionary access control* (D-AC), whereby it is at the resource owner's discretion what permissions to grant others regarding that resource. In contrast, in *mandatory access control* (M-AC) systems, a security policy administrator defines, for every object (resource), which subjects have which permissions on it. One type of M-AC system, the *multi-level security* (MLS) model of the U.S. government (Dept. of Defense), assigns to each user (subject) a security clearance level, and correspondingly classifies documents:

(Top Secret, Secret, Confidential, Controlled Unclassified, Unclassified)

A subject may access a document if their clearance is equal to or exceeds the document's classification. For this type of M-AC, and D-AC, the basis on which permissions are assigned is commonly subject identity (userid). In the system discussed next, permissions are assigned based on user *roles*, in what is neither purely a M-AC nor a D-AC system (and can possibly support either).

RBAC. The idea of *role-based access control* (Figure 5.10) is that a user, represented as a *subject*, is assigned one or more *roles* in each active session. Each role is pre-assigned a set of permissions. A subject's current roles then determine its permissions. This reflects how permissions are often assigned in larger organizations (enterprises).

Example *(RBAC).* Suppose the role *GradAdmin* is assigned read and write access to department files related to current students, new applicants, and office supply budgets; and

[12]Misunderstanding file deletion has implications for principle P16 (REMNANT-REMOVAL). Also, when a filesystem itself releases a file, file memory is typically not overwritten, i.e., *secure deletion* is not guaranteed.

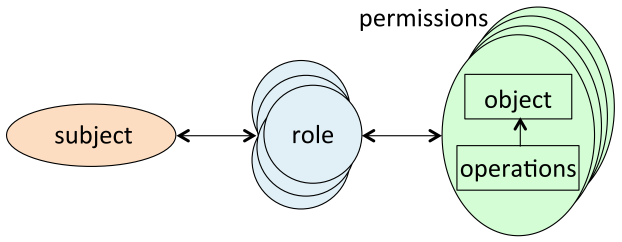

Figure 5.10: Role-based access control (RBAC) model. A user, represented as a subject, is pre-assigned one or more roles by an administrator. The administrator also assigns permissions to each role. For each login session, a subset of roles is activated for the user.

the role *GrantManager* is assigned read access to files for department member grants. A new staff member Alex who is assigned both these roles will then acquire both sets of permissions. When Alex moves to another department and Corey takes over, Corey gets the same permissions by being assigned these two roles; if individual file-based permissions were used, a longer list of individual permissions might have to be reassigned. Roles may be hierarchically defined, e.g., so that a *SeniorManager* role is the union of all roles enjoyed by junior managers, plus some roles specific to the higher position. RBAC system administrators make design choices as to which tasks (and corresponding permissions) are associated with different job functions, and define roles accordingly.

M-AC AND SELINUX. The remainder of this section introduces efforts related to SELinux: the Flask operating system architecture on which it was built, the Linux Security Module framework that provides support for it (and other M-AC approaches) within Linux, and the SEAndroid version of it for the Google Android OS.

‡**Exercise** (Flask). The Flask security architecture was designed as a prototype during the 1990s to demonstrate that a commodity OS could support a wide range of (mandatory) access control policies. a) Summarize the motivation and goals of Flask. b) Describe the Flask architecture, including a diagram showing how its Client, Object Manager, and Security Server interact. c) Explain Flask *object labeling* (include a diagram), and how the data types *security context* and *security identifier* (SID) fit in. (Hint: [34].)

‡**Exercise** (SELinux). Security-Enhanced Linux (SELinux) is a modified version of Linux built on the Flask architecture and its use of *security labels*. SELinux supports mandatory security policies and enforcement of M-AC policies across all processes. It was originally integrated into Linux as a kernel patch, and reimplemented as an LSM (below). a) Summarize details of the SELinux implementation of the Flask architecture, including the role security labels play and how they are supported. b) Describe the security mechanisms provided by SELinux, including mandatory access controls for process management, filesystem objects, and sockets. c) Describe the SELinux API. d) Give an overview of the SELinux example security policy configuration that serves as a customizable foundation from which to build other policy specifications. (Hint: [22].)

‡**Exercise** (LSMs). Linux Security Modules (LSMs) are a general framework by which the Linux kernel can support, and enforce, diverse advanced access control approaches including SELinux. This is done by exposing kernel abstractions and operations

to an LSM; different modules can then implement distinct approaches. a) Describe in greater detail the general problem that the LSM project addresses. b) Summarize the technical details of the LSM design. (Hint: [40].)

‡**Exercise** (SEAndroid). Security-Enhanced Android (SEAndroid) brings mandatory access control to Google's Android OS, through an Android kernel that supports SELinux. a) Summarize the main challenges in supporting SELinux on Android. b) Summarize the technical means by which these challenges were overcome. c) Summarize the security benefits offered by SEAndroid. (Hint: [33].)

5.8 ‡Protection rings: isolation meets finer-grained sharing

Practical requirements call for an efficient middleground between the security of full isolation (complete containment, no sharing) and the convenience of shared objects. Protection rings generalize the (supervisor, user mode) hardware model to multiple privilege levels and domains. They also offer concrete examples for principles P4 (COMPLETE-MEDIATION), P5 (ISOLATED-COMPARTMENTS) and P7 (MODULAR-DESIGN).

PROTECTION RINGS. Section 5.1 introduced isolation, and selective (basic) sharing across processes. A third desirable feature is layered protection within processes—affording user-space processes separation analogous to supervisor-user separation, and sharing of *protected subsystems*. This can be provided by *protection rings*, introduced by Multics in the mid-1960s including eventual hardware support for eight rings. Rings overlay additional access control on Multics segmented memory, and generalize privilege classes from two (supervisor, user) to n. Rings selectively allow complete isolation of processes, controlled sharing between programs (e.g., for reuse of common code and data), and layered protection for varying degrees of separation. The idea is that segments in stronger rings are protected from access by weaker rings; these conditions are then relaxed to provide greater flexibility, when authorized.

RING NUMBERS AND SUPPORT. Consider a set of rings 0 to $n-1$ as a nested, ordered set of levels, per Fig. 5.11a. Ring 0 (center) is the most privileged. Privilege decreases moving outward—to aid memory, think "the core is strong". As low ring numbers

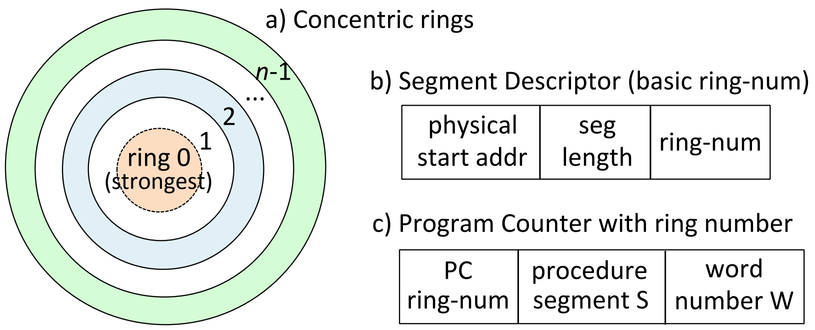

Figure 5.11: Protection rings and supporting descriptors. To support protection rings, ring numbers are added to segment descriptors and to the program counter.

having high privileges may be confusing, we also use the terms *stronger* (*inner*) rings and *weaker* (*outer*) rings. We add to every segment descriptor (Section 5.1) a new `ring-num` field (ring number) per Figure 5.11b, and to the CPU Program Counter (instruction address register) a `PCring-num` for the ring number of the executing process, per Fig. 5.11c. For access control functionality, we associate with each segment an *access bracket* (n_1, n_2) denoting a range of consecutive rings, used as explained below.

PROCEDURE SEGMENT ACCESS BRACKETS. A user process may desire to transfer control to programs designed to execute in stronger rings (e.g., for privileged functions, such as input-output functions, or to change permissions to a segment's access control list in supervisor memory); or weaker rings (e.g., for simple shared services). A user process P_1 may wish to allow another user's process P_2, operating in a weaker ring, access to data memory in P_1's ring, but only under the condition that such access is through a program (segment) provided by P_1, and verified to have been accessed at a pre-authorized *entry point*, specified by a memory address. Rings allow this, but now transfers that change rings (*cross-ring transfers*) require extra checks. In the simple "within-bracket" case, a calling process P_1 executing in ring i requests a transfer to procedure segment P_2 with *access bracket* (n_1, n_2) and $n_1 \leq i \leq n_2$. Transfer is allowed, without any change of control ring (`PCring-num` remains i).[13] The more complicated "out-of-bracket" case is when i is outside P_2's access bracket. Such transfer requests trigger a fault; control goes to the supervisor to sort out, as discussed next.

PROCEDURE SEGMENT GATE EXTENSION. Suppose a process in ring $i > n_2$ attempts transfer to a stronger ring bracketed (n_1, n_2). Processes are not generally permitted to call stronger-ring programs, but to allow flexibility, the design includes a parameter, n_3, designating a *gate extension* for a triple (n_1, n_2, n_3). For case $n_2 < i \leq n_3$, a transfer request is now allowed, but only to pre-specified entry points. A list (*gate list*) of such entry points is specified by any procedure segment to be reachable by this means. So $i > n_2$ triggers a fault and a software fault handler handles case $n_2 < i \leq n_3$. The imagery is that gates are needed to cross rings, mediated by *gatekeeper* software as summarized next.

RING GATEKEEPER MEDIATION. When a ring-i process P_1 requests transfer to procedure segment P_2 having ring bracket (n_1, n_2, n_3), a mediation occurs per Table 5.2.

Case	Meaning	Action
$n_1 \leq i \leq n_2$	within access bracket	allow transfer†
$i < n_1$ or $n_2 < i$	triggers fault for gatekeeper resolution (based on subcase)	
subcase: $i < n_1$	calling weaker ring	allow transfer†
subcase: $n_2 < i \leq n_3$	within gate extension	allow if transfer address on gatelist
subcase: $n_3 < i$	above gate extension	error, unauthorized transfer

Table 5.2: Mediation by ring gatekeeper. †On transfer to a weaker ring P_2, the gatekeeper should check that all arguments passed will be accessible (P_2 may be unable to access higher-privilege memory); one option is to copy arguments into accessible memory.

[13]It would be unclear which value in the access bracket (n_1, n_2) to change the `PCring-num` to, as the bracket declares the program suitable in the full range. This suggests that many programs might naturally designate a single-ring access bracket, namely the ring best suiting the program.

Cross-ring returns. Cross-ring returns (e.g., P_2 returning to P_1) likewise trigger mediation, and may involve *return gates*, which we do not discuss further. The gatekeeper enforces that returns match stack expectations, using details stored on the standard call stack such as segment descriptors of return segments (including previous `ring-num`).

Ring number after transfer from outside-bracket. After an outside-bracket transfer into bracket (n_1, n_2), what value x should be assigned to `PCring-num`? The easy case is $n_2 < i \leq n_3$: LEAST-PRIVILEGE (P6) suggests $x = n_2$, temporarily increasing privileges by the least necessary. For $i < n_1$, privileges should be reduced, with strict least-privilege dictating $x = n_2$, while the "fewest-hops" choice $x = n_1$ may be slightly more compelling for overall simplicity, given that an "in-bracket" transfer from ring n_2 would leave the ring of execution at n_2. Thus "fewest-hops" provides a reasonable choice for x in both cases. Another choice would be for the segment, including possibly its gate entry, to specify a new ring of execution.

Exercise (Address arguments passed to stronger segment). Suppose weaker program segment P_w calls stronger program segment P_s, passing an argument involving a memory address A that P_s is sufficiently privileged to access, but which P_w is not. P_w does not itself try to access A (so there is no access fault). Is it possible that P_s could, as a result, disrupt the integrity of its own data segment, or damage other segments in its ring? Are additional gatekeeper actions, therefore, necessary for inward calls? If so, explain. (Hint: [31].)

Execute, read and write brackets. We now consider *read access brackets* and *write access brackets*, with semantics as explained below. Our discussion of cross-ring execution privileges involved access bracket and gate extension parameters (n_1, n_2, n_3). These values might be stored in hardware registers R_1, R_2, R_3 respectively, populated from corresponding values in Multics segment descriptors, defining an *execute bracket* (R_1, R_2) and gate extension $(R_2 + 1, R_3)$. To define corresponding read and write access brackets respectively delimited by integer pairs (d_1, d_2) and (w_1, w_2), it would be convenient to reuse the same registers. Consider these choices (explained next), per Figure 5.12a: read bracket $(0, R_2)$, write bracket $(0, R_1)$, with $R_1 \leq R_2 \leq R_3$.

Reasoning for choices. The read and write lower bounds stem from reasoning that ring 0 processes should have access if any less privileged rings do. Equating the write bracket top and execute bracket bottom allows a single ring, R1, in which a segment can be both written and executed (arguably, ruling out some possible programming errors while retaining flexibility). Setting equal the read and execute bracket tops appears reasonable (to deny R access, turn the R bit off). This provides a complete ruleset for RWX access to a segment. Access requires both (1) the requesting process' ring be within the segment's bracket, and (2) the segment descriptor's relevant RWX flag be on. See Figure 5.12b.

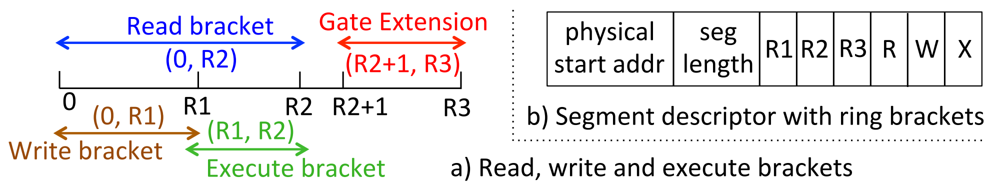

Figure 5.12: a) Brackets defined by three registers. b) Descriptor with ring registers.

ALTERNATE BRACKET RULESET. For a data segment D implemented using the same registers R1, R2, R3 as above, suppose parameters (n_1, n_2) are interpreted differently, to define the following alternative ruleset, for a process running in ring i:

1) write bracket $(0, n_1)$: can write into D if $i \leq n_1$ (and D's descriptor allows W)
2) read bracket $(n_1 + 1, n_2)$: can read D if $n_1 < i \leq n_2$ (and D's descriptor allows R)

For example then, for $(n_1, n_2) = (2, 3)$, a ring-i process has access as follows: for $i = 1, 2$, can only write to D; for $i = 3$, can only read from D; and for $i = 4$, can neither read nor write D. By this ruleset, in no case can the ring-i process both read from and write to D; to do both, a process would have to alternate between rings.

Exercise (Ruleset discussion). Is the alternate bracket ruleset useful, i.e., does it offer advantages over other possible rulesets? (Hint: [14]. Note that a ruleset has implications for subsystem design, e.g., in which rings to locate program functionality. Different permissions can be specified in different segment descriptors, but a single ruleset is fixed for all processes.)

SEGMENT DESCRIPTORS FROM ACCESS CONTROL ENTRIES. To complete the Multics story of access control through segment descriptors, consider how these themselves originate. When a user logs in, a new process P is created for the user activity. The virtual memory that P can see—its universe of addressable memory—is limited to memory reachable by the segment descriptors in P's descriptor segment. The supervisor creates P's virtual memory space by populating this descriptor segment. Recall that the descriptor base register (DBR, Fig. 5.2) points to the base of P's descriptor segment table. How does the supervisor select which segments to give P descriptors for? Segments (as objects) have corresponding ACL entries in the system, specifying which subjects may access them. From such entries, the supervisor constructs P's descriptor segment.

Exercise (Supervisor creation of descriptor segment). A user logs in to their account. The supervisor creates a new process P for the user activity, and sets out to create the descriptor segment for P. Abstractly, there is an access control matrix with subjects as rows, and objects (including segments) as columns. a) Discuss which implementation more efficiently supports the supervisor task of creating P's descriptor segment: a matrix stored by row in the form of capabilities, or by column in the form of ACLs. b) Where does the supervisor find information from which to appropriately populate the access control indicators necessary in segment descriptors? (Hint: [13].)

5.9 ‡Relating subjects, processes, and protection domains

Protections rings are an example of *protection domains*, sometimes also called protection *contexts* or protection *environments*. Here we define domains and subjects more precisely.

SUBJECTS AND DOMAINS. We first refine our definition of *subject*, a term used as row index of the access matrix in the subject-object permission model (Section 5.2). Access control controls access to objects, requested by subjects. We define a *subject* S as a tuple including a *process* $\mathcal{P}$ executing on its behalf, and a *domain* $\mathcal{D}$ (explained next):

$$S = (\mathcal{P}, \mathcal{D})$$

The domain of a process—introduced in Section 5.1 as the permission set associated with the objects a process can access—can change over time. When a Unix process calls a root-owned setuid program it retains its process identifier (PID), but temporarily runs with a different `eUID`, acquiring different access privileges. Viewing the domain as a room, the objects in the room are accessible to the subject; when the subject changes rooms, the accessible objects change. For a more precise definition of *protection domain* $\mathcal{D}$, the ring system and segmented virtual addressing of Multics can be used as a basis. We define the domain of a subject $S = (\mathcal{P}, \mathcal{D})$ associated with process $\mathcal{P}$ as

$$\mathcal{D} = (r, T)$$

Here r is the execution ring of the segment g that $\mathcal{P}$ is running in, and T is $\mathcal{P}$'s descriptor segment table containing segment descriptors (including for g), each including access indicators. This yields a more detailed description of a subject as

$$S = (\mathcal{P}, r, T) = (\texttt{processID, ring-number, descriptor-seg-ptr})$$

NOTES. These definitions for subject and domain lead to the following observations.

1) A change of execution ring changes the domain, and thus the privileges associated with a subject S. At any specific execution point, a process operates in one domain or context; privileges change with context (mode or ring).
2) A transfer of control to a different segment, but within the same ring (same process, same descriptor segment), changes neither the domain nor the subject.
3) Access bracket entry points specify allowed, gatekeeper-enforced domain changes.
4) Virtual address translation maps constrain physical memory accessible to a domain.
5) A system with n protection rings defines a strictly ordered set of n protection domains, $\mathcal{D}^{(i)} = (i, T), 0 \leq i \leq n-1$. Associating these with a process $\mathcal{P}$ and its fixed descriptor segment defined by T, defines a set of subjects $(\mathcal{P}, 0, T), ..., (\mathcal{P}, n-1, T)$.
6) Informally, C-lists (Section 5.2) define the environment of a process. Equating C-lists with domains allows substitution in the definition $S = (\mathcal{P}, \mathcal{D})$.

Exercise (Ring changes vs. switching userid). It is recommended that distinct accounts be set up on commodity computers to separate regular user and administrative activities. Tasks requiring superuser privileges employ the administrative account. Discuss how this compares, from an OS viewpoint, to a process changing domains by changing rings.

HARDWARE-SUPPORTED CPU MODES UNUSED BY SOFTWARE. Many computers in use run operating systems supporting only two CPU modes (supervisor, user) despite hardware support for more—thus available hardware support for rings goes unused (Figure 5.13). Why so? If an OS vendor seeks maximum market share via deployment on all plausible hardware platforms, the lowest-functionality hardware (here, in terms of CPU modes) constrains all others. The choices are to abandon deployment on the low-functioning hardware (ceding market share), incur major costs of redesign and support for multiple software streams across hardware platforms, or reduce software functionality across all platforms. Operating systems custom-built for specific hardware can offer richer features, but fewer hardware platforms are then compatible for deployment.

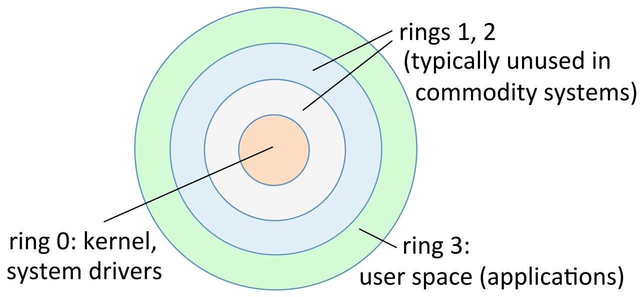

Figure 5.13: Underuse of hardware-supported protection rings. Widely used Intel x86 hardware supports four modes (rings); Windows systems running on it use rings 0 and 3. For example, device drivers might be put in ring 1, albeit adding context-switching costs.

Exercise (Platforms supporting more than two modes). Discuss, with technical details, how many CPU modes, privilege levels, or rings are supported by the ARMv7 processor architecture. Likewise discuss for operating systems OpenVMS and OS/2. Do any versions of Windows support more than two modes?

5.10 ‡End notes and further reading

Jaeger [16] is recommended as a recent short book covering OS security, SELinux, security kernels, and Multics including rings directly supporting multi-level security (MLS) policies. Curry [4] addresses Unix security concisely. Ritchie [26] gives an early overview of Unix. Recommended books on *operating systems* include Saltzer and Kaashoek [28] (for security *design principles*), Silberschatz [32, Chapter 14-15], and Tanenbaum [35] for conciseness and clarity. See Watson [39] for *Capsicum capabilities*, which extend Unix file descriptors, and CHERI [38] for hardware-software support of capability-based memory protection. Gruenbacher [15] summarizes *ACL* support in Unix-like systems; for the Multics origin of ACLs and hierarchical directory files, see Daley [5]. Dittmer [7], and earlier Chen [3], explore inconsistent implementations of setuid() and related system calls; see also Dowd [8, Chapter 9]. Loscocco [23] discusses limitations of *D-AC* (largely those that motivated SELinux), and argues for renewed interest in secure operating systems, as necessary to support computer security in general. For SELinux, see McCarty [24]. For *RBAC*, see Ferraiolo [9], Sandhu [30], and the RBAC reference model [10]. RBAC supports [9] the principles of LEAST-PRIVILEGE P6, MODULAR-DESIGN P7 and related separation of duty. For better compartmentalization than available via chroot(), Kamp [17] introduced jail() to FreeBSD; Bellovin [2, §10.3] offers thoughts on the more general concept of *sandboxing*.

Lampson's 1971 conference paper [19] unified early access control mechanisms under the *access matrix model*. Section 5.1 draws from Graham [14] and Saltzer and Schroeder [29] (see also for *capabilities*). For an insightful short Multics survey including security features omitted herein, see Saltzer [27]; for a comprehensive technical exposition of the early-Multics design plans with details of internal mechanisms, see Organick [25].

Dennis [6] is credited for *segmented addressing* (although protection features of Multics-style segmentation were designed out of later commercial systems). Graham [14] gives an authoritative view of *protection rings* (including early identification of *race condition issues*); see also Graham and Denning [13] (which Section 5.9 follows), and Schroeder and Saltzer [31] for a Multics-specific discussion of hardware-supported rings and related software issues. Interest in using hardware-supported rings recurs—for example, Lee [20] leverages unused x86 rings for portable user-space privilege separation.

The 1970 Ware report [37] explored security controls in resource-sharing computer systems. The 1972 Anderson report [1, pages 8-14] lays out the central ideas of the *reference monitor* concept, access matrix, and security kernel; it expressed early concerns about requiring trust in the entire computer-manufacturing supply chain, and the ability to determine that *"compiler and linkage editors are either certified free from 'trap-doors', or that their output can be checked and verified independently"*—attacks later more fully explained in Thompson's Turing-award paper [36] detailing C-code for a Trojan horse compiler. The 1976 RISOS report [21, p.57] defined a *security kernel* as *"that portion of an operating system whose operation must be correct in order to ensure the security of the operating system"* (cf. Chapter 1, principle of SMALL-TRUSTED-BASES P8). Their small-size requirement, originally specified as part of the validation mechanism for the reference monitor, has made *microkernels* a focus for security kernels (cf. Jaeger above). These 1970-era reports indicate longstanding awareness of computer security challenges.

Lampson's 1973 note on the *confinement problem* [18] raises the subject of untrusted programs leaking data over *covert channels*. Gasser's lucid 1988 book [12], a practitioner's integrated treatment on how to build secure computer systems, includes discussion of reference monitors, security kernels, segmented virtual memory, MLS/mandatory access control, and covert channels.

References (Chapter 5)

[1] J. P. Anderson. Computer Security Technology Planning Study (Vol. I and II, “Anderson report”). James P. Anderson and Co., Fort Washington, PA, USA, Oct 1972.

[2] S. M. Bellovin. *Thinking Security: Stopping Next Year’s Hackers*. Addison-Wesley, 2016.

[3] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *USENIX Security*, 2002.

[4] D. A. Curry. *UNIX System Security: A Guide for Users and System Administrators*. Addison-Wesley, 1992.

[5] R. C. Daley and P. G. Neumann. A general-purpose file system for secondary storage. In *AFIPS Fall Joint Computer Conference*, pages 213–229, Nov 1965.

[6] J. B. Dennis. Segmentation and the design of multiprogrammed computer systems. *Journal of the ACM*, 12(4):589–602, 1965.

[7] M. S. Dittmer and M. V. Tripunitara. The UNIX process identity crisis: A standards-driven approach to setuid. In *ACM Comp. & Comm. Security (CCS)*, pages 1391–1402, 2014.

[8] M. Dowd, J. McDonald, and J. Schuh. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley, 2006.

[9] D. Ferraiolo and R. Kuhn. Role-based access controls. In *National Computer Security Conf. (NCSC)*, pages 554–563, Oct. 1992.

[10] D. F. Ferraiolo, R. S. Sandhu, S. I. Gavrila, D. R. Kuhn, and R. Chandramouli. Proposed NIST standard for role-based access control. *ACM Trans. Inf. Systems and Security*, 4(3):224–274, 2001.

[11] W. Ford and M. J. Wiener. A key distribution method for object-based protection. In *ACM Comp. & Comm. Security (CCS)*, pages 193–197, 1994.

[12] M. Gasser. *Building a Secure Computer System*. Van Nostrand Reinhold, 1988. PDF available online.

[13] G. S. Graham and P. J. Denning. Protection—principles and practice. In *AFIPS Spring Joint Computer Conference*, pages 417–429, May 1972.

[14] R. M. Graham. Protection in an information processing utility. *Comm. ACM*, 11(5):365–369, 1968. Appeared as the first paper, pp.1-5, first ACM Symposium on Operating System Principles, 1967.

[15] A. Gruenbacher. POSIX access control lists on LINUX. In *USENIX Annual Technical Conf.*, pages 259–272, 2003.

[16] T. Jaeger. *Operating System Security*. Morgan and Claypool, 2008.

[17] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root. In *System Admin. and Networking Conf. (SANE)*, 2000. Cf. “Building systems to be shared, securely”, *ACM Queue*, Aug 2004.

[18] B. W. Lampson. A note on the confinement problem. *Comm. ACM*, 16(10):613–615, 1973.

[19] B. W. Lampson. Protection. *ACM Operating Sys. Review*, 8(1):18–24, 1974. Originally published in *Proc. 5th Princeton Conf. on Information Sciences and Systems*, 1971.

[20] H. Lee, C. Song, and B. B. Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *ACM Comp. & Comm. Security (CCS)*, pages 1441–1454, 2018.

[21] T. Linden. Security Analysis and Enhancements of Computer Operating Systems (“RISOS report”), Apr 1976. NBSIR 76-1041, The RISOS Project, Lawrence Livermore Laboratory, Livermore, CA.

[22] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *USENIX Annual Technical Conf.*, pages 29–42, 2001. FREENIX Track. Full technical report, 62 pages, available online.

[23] P. A. Loscocco, S. D. Smalley, P. A. Muckelbauer, R. C. Taylor, S. J. Turner, and J. F. Farrell. The inevitability of failure: The flawed assumption of security in modern computing environments. In *National Info. Systems Security Conf. (NISSC)*, pages 303–314, 1998.

[24] B. McCarty. *SELinux: NSA’s Open Source Security Enhanced Linux*. O’Reilly Media, 2004.

[25] E. I. Organick. *The Multics System: An Examination of Its Structure*. MIT Press (5th printing, 1985), 1972.

[26] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Comm. ACM*, 17(7):365–375, 1974.

[27] J. H. Saltzer. Protection and the control of information sharing in Multics. *Comm. ACM*, 17(7):388–402, 1974.

[28] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design*. Morgan Kaufmann, 2010.

[29] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, September 1975.

[30] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[31] M. D. Schroeder and J. H. Saltzer. A hardware architecture for implementing protection rings. *Comm. ACM*, 15(3):157–170, 1972. Earlier version in *ACM SOSP*, pages 42–54, 1971.

[32] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts (seventh edition)*. John Wiley and Sons, 2005.

[33] S. Smalley and R. Craig. Security Enhanced (SE) Android: Bringing Flexible MAC to Android. In *Netw. Dist. Sys. Security (NDSS)*, 2013.

[34] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *USENIX Security*, 1999.

[35] A. S. Tanenbaum. *Modern Operating Systems (3rd edition)*. Pearson Prentice Hall, 2008.

[36] K. Thompson. Reflections on trusting trust. *Comm. ACM*, 27(8):761–763, 1984.

[37] W. H. Ware (Chair). Security Controls for Computer Systems: Report of Defense Science Board Task Force on Computer Security. RAND Report R-609-1 (“Ware report”), 11 Feb 1970. Office of Director of Defense Research and Engineering, Wash., D.C. Confidential; declassified 10 Oct 1975.

[38] R. N. M. Watson and 14 others. CHERI: A hybrid capability-system architecture for scalable software compartmentalization. In *IEEE Symp. Security and Privacy*, pages 20–37, 2015.

[39] R. N. M. Watson, J. Anderson, B. Laurie, and K. Kennaway. Capsicum: Practical capabilities for UNIX. In *USENIX Security*, pages 29–46, 2010.

[40] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security*, pages 17–31, 2002.