We need to provide protection for several reasons. The most obvious is the need to prevent the mischievous, intentional violation of an access restriction by a user. Of more general importance, however, is the need to ensure that each process in a system uses system resources only in ways consistent with stated policies. This requirement is an absolute one for a reliable system.

Protection can improve reliability by detecting latent errors at the interfaces between component subsystems. Early detection of interface errors can often prevent contamination of a healthy subsystem by a malfunctioning subsystem. Also, an unprotected resource cannot defend against use (or misuse) by an unauthorized or incompetent user. A protection-oriented system provides means to distinguish between authorized and unauthorized usage.

The role of protection in a computer system is to provide a mechanism for the enforcement of the policies governing resource use. These policies can be established in a variety of ways. Some are fixed in the design of the system, while others are formulated by the management of a system. Still others are defined by individual users to protect resources they "own." A protection system, then, must have the flexibility to enforce a variety of policies.

Policies for resource use may vary by application, and they may change over time. For these reasons, protection is no longer the concern solely of the designer of an operating system. The application programmer needs to use protection mechanisms as well, to guard resources created and supported by an application subsystem against misuse. In this chapter, we describe the protection mechanisms the operating system should provide, but application designers can use them as well in designing their own protection software.

Note that ***mechanisms*** are distinct from ***policies.*** Mechanisms determine ***how*** something will be done; policies decide ***what*** will be done. The separation of policy and mechanism is important for flexibility. Policies are likely to change from place to place or time to time. In the worst case, every change in policy would require a change in the underlying mechanism. Using general mechanisms enables us to avoid such a situation.

17.2 Principles of Protection

Frequently, a guiding principle can be used throughout a project, such as the design of an operating system. Following this principle simplifies design decisions and keeps the system consistent and easy to understand. A key, time-tested guiding principle for protection is the **principle of least privilege**. As discussed in Chapter 16, this principle dictates that programs, users, and even systems be given just enough privileges to perform their tasks.

Consider one of the tenets of UNIX—that a user should not run as root. (In UNIX, only the root user can execute privileged commands.) Most users innately respect that, fearing an accidental delete operation for which there is no corresponding undelete. Because root is virtually omnipotent, the potential for human error when a user acts as root is grave, and its consequences far reaching.

Now consider that rather than human error, damage may result from malicious attack. A virus launched by an accidental click on an attachment is one example. Another is a buffer overflow or other code-injection attack that is successfully carried out against a root-privileged process (or, in Windows,

a process with administrator privileges). Either case could prove catastrophic for the system.

Observing the principle of least privilege would give the system a chance to mitigate the attack—if malicious code cannot obtain root privileges, there is a chance that adequately defined **permissions** may block all, or at least some, of the damaging operations. In this sense, permissions can act like an immune system at the operating-system level.

The principle of least privilege takes many forms, which we examine in more detail later in the chapter. Another important principle, often seen as a derivative of the principle of least privilege, is **compartmentalization**. Compartmentalization is the process of protecting each individual system component through the use of specific permissions and access restrictions. Then, if a component is subverted, another line of defense will "kick in" and keep the attacker from compromising the system any further. Compartmentalization is implemented in many forms—from network demilitarized zones (DMZs) through virtualization.

The careful use of access restrictions can help make a system more secure and can also be beneficial in producing an **audit trail**, which tracks divergences from allowed accesses. An audit trail is a hard record in the system logs. If monitored closely, it can reveal early warnings of an attack or (if its integrity is maintained despite an attack) provide clues as to which attack vectors were used, as well as accurately assess the damage caused.

Perhaps most importantly, no single principle is a panacea for security vulnerabilities. **Defense in depth** must be used: multiple layers of protection should be applied one on top of the other (think of a castle with a garrison, a wall, and a moat to protect it). At the same time, of course, attackers use multiple means to bypass defense in depth, resulting in an ever-escalating arms race.

17.3 Protection Rings

As we've seen, the main component of modern operating systems is the kernel, which manages access to system resources and hardware. The kernel, by definition, is a trusted and privileged component and therefore must run with a higher level of privileges than user processes.

To carry out this ***privilege separation,*** hardware support is required. Indeed, all modern hardware supports the notion of separate execution levels, though implementations vary somewhat. A popular model of privilege separation is that of protection rings. In this model, fashioned after Bell–LaPadula (https://www.acsac.org/2005/papers/Bell.pdf), execution is defined as a set of concentric rings, with ring i providing a subset of the functionality of ring j for any $j < i$. The innermost ring, ring 0, thus provides the full set of privileges. This pattern is shown in Figure 17.1.

When the system boots, it boots to the highest privilege level. Code at that level performs necessary initialization before dropping to a less privileged level. In order to return to a higher privilege level, code usually calls a special instruction, sometimes referred to as a gate, which provides a portal between rings. The `syscall` instruction (in Intel) is one example. Calling this instruction shifts execution from user to kernel mode. As we have seen, executing a system