CHAPTER 1

# Introduction

Operating systems are the software that provides access to the various hardware resources (e.g., CPU, memory, and devices) that comprise a computer system as shown in Figure 1.1. Any program that is run on a computer system has instructions executed by that computer's CPU, but these programs may also require the use of other peripheral resources of these complex systems. Consider a program that allows a user to enter her password. The operating system provides access to the disk device on which the program is stored, access to device memory to load the program so that it may be executed, the display device to show the user how to enter her password, and keyboard and mouse devices for the user to enter her password. Of course, there are now a multitude of such devices that can be used seamlessly, for the most part, thanks to the function of operating systems.

As shown in Figure 1.1, operating systems run programs in *processes*. The challenge for an operating system developer is to permit multiple concurrently executing processes to use these resources in a manner that preserves the independence of these processes while providing fair sharing of these resources. Originally, operating systems only permitted one process to be run at a time (e.g., *batch systems*), but as early as 1960, it became apparent that computer productivity would be greatly enhanced by being able to run multiple processes concurrently [87]. By *concurrently*, we mean that while only one process uses a computer's CPU at a time, multiple other processes may be in various states of execution at the same time, and the operating system must ensure that these executions are performed effectively. For example, while the computer waits for a user to enter her password, other processes may be run and access system devices as well, such as the network. These systems were originally called *timesharing systems*, but they are our default operating systems today.

To build any successful operating system, we identify three major tasks. First, the operating system must provide various mechanisms that enable high performance use of computer resources. Operating systems must provide efficient *resource mechanisms*, such as file systems, memory management systems, network protocol stacks, etc., that define how processes use the hardware resources. Second, it is the operating system's responsibility to switch among the processes fairly, such that the user experiences good performance from each process in concert with access to the computer's devices. This second problem is one of *scheduling* access to computer resources. Third, access to resources should be controlled, such that one process cannot inadvertently or maliciously impact the execution of another. This third problem is the problem of ensuring the *security* of all processes run on the system.

Ensuring the secure execution of all processes depends on the correct implementation of resource and scheduling mechanisms. First, any correct resource mechanism must provide boundaries between its objects and ensure that its operations do not interfere with one another. For example, a file system must not allow a process request to access one file to overwrite the disk space allocated

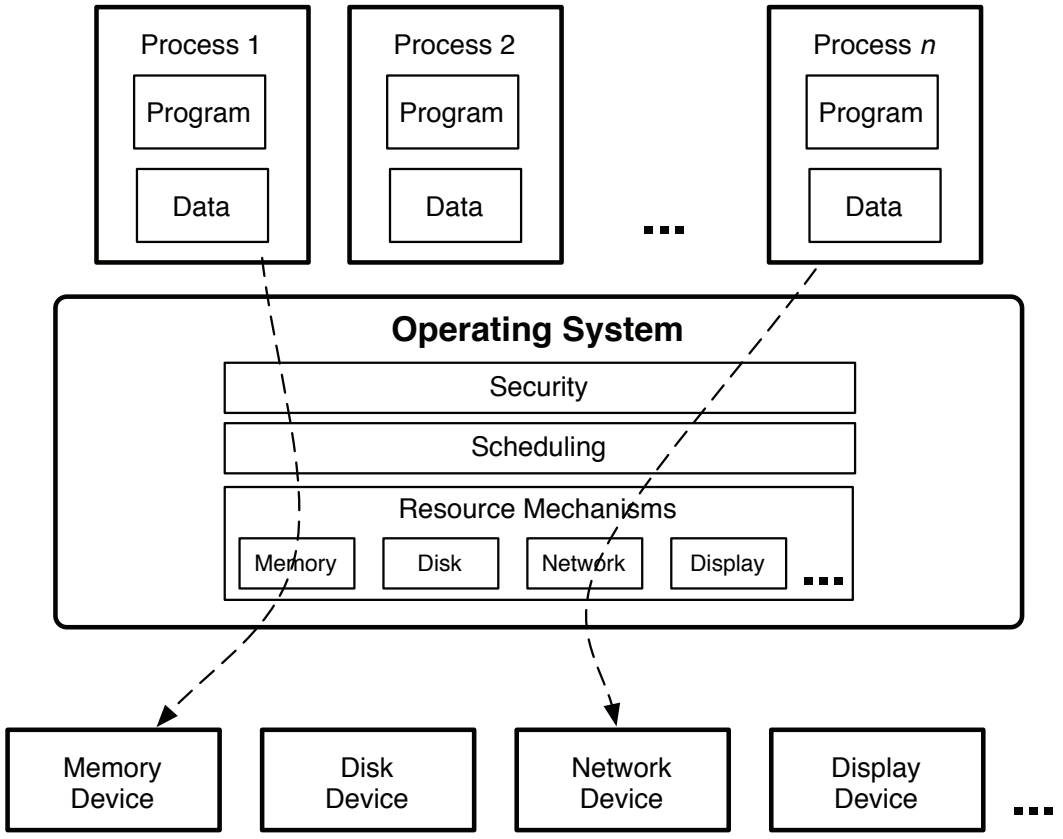


**Figure 1.1:** An operating system runs *security*, *scheduling*, and *resource mechanisms* to provide *processes* with access to the computer system's resources (e.g., CPU, memory, and devices).

to another file. Also, file systems must ensure that one write operation is not impacted by the data being read or written in another operation. Second, scheduling mechanisms must ensure availability of resources to processes to prevent denial of service attacks. For example, the algorithms applied by scheduling mechanisms must ensure that all processes are eventually scheduled for execution. These requirements are fundamental to operating system mechanisms, and are assumed to be provided in the context of this book. The scope of this book covers the misuse of these mechanisms to inadvertently or, especially, maliciously impact the execution of another process.

Security becomes an issue because processes in modern computer systems interact in a variety of ways, and the sharing of data among users is a fundamental use of computer systems. First, the output of one process may be used by other processes. For example, a programmer uses an editor program to write a computer program's source code, compilers and linkers to transform the program

code into a form in which it can be executed, and debuggers to view the executing processes image to find errors in source code. In addition, a major use of computer systems is to share information with other users. With the ubiquity of Internet-scale sharing mechanisms, such as e-mail, the web, and instant messaging, users may share anything with anyone in the world. Unfortunately, lots of people, or at least lots of email addresses, web sites, and network requests, want to share stuff with you that aims to circumvent operating system security mechanisms and cause your computer to share additional, unexpected resources. The ease with which malware can be conveyed and the variety of ways that users and their processes may be tricked into running malware present modern operating system developers with significant challenges in ensuring the security of their system's execution.

The challenge in developing operating systems security is to design security mechanisms that protect process execution and their generated data in an environment with such complex interactions. As we will see, formal security mechanisms that enforce provable security goals have been defined, but these mechanisms do not account or only partially account for the complexity of practical systems. As such, the current state of operating systems security takes two forms: (1) constrained systems that can enforce security goals with a high degree of assurance and (2) general-purpose systems that can enforce limited security goals with a low to medium degree of assurance. First, several systems have been developed over the years that have been carefully crafted to ensure correct (i.e., within some low tolerance for bugs) enforcement of specific security goals. These systems generally support few applications, and these applications often have limited functionality and lower performance requirements. That is, in these systems, security is the top priority, and this focus enables the system developers to write software that approaches the ideal of the formal security mechanisms mentioned above. Second, the computing community at large has focused on function and flexibility, resulting in general-purpose, extensible systems that are very difficult to secure. Such systems are crafted to simplify development and deployment while achieving high performance, and their applications are built to be feature-rich and easy to use. Such systems present several challenges to security practitioners, such as insecure interfaces, dependence of security on arbitrary software, complex interaction with untrusted parties anywhere in the world, etc. But, these systems have defined how the user community works with computers. As a result, the security community faces a difficult task for ensuring security goals in such an environment.

However, recent advances are improving both the utility of the constrained systems and the security of the general-purpose systems. We are encouraged by this movement, which is motivated by the general need for security in all systems, and this book aims to capture many of the efforts in building security into operating systems, both constrained and general-purpose systems, with the aim of enabling broader deployment and use of security function in future operating systems.

## 1.1 SECURE OPERATING SYSTEMS

The ideal goal of operating system security is the development of a secure operating system. *A secure operating system provides security mechanisms that ensure that the system's security goals are enforced despite the threats faced by the system.* These security mechanisms are designed to provide such a guarantee in

the context of the resource and scheduling mechanisms. Security goals define the requirements of secure operation for a system for any processes that it may execute. The security mechanisms must ensure these goals regardless of the possible ways that the system may be misused (i.e., is threatened) by attackers.

The term "secure operating system" is both considered an ideal and an oxymoron. Systems that provide a high degree of assurance in enforcement have been called secure systems, or even more frequently "trusted" systems [1]. However, it is also true that no system of modern complexity is completely secure. The difficulty of preventing errors in programming and the challenges of trying to remove such errors means that no system as complex as an operating system can be completely secure.

Nonetheless, we believe that studying how to build an ideal secure operating system to be useful in assessing operating systems security. In Chapter 2, we develop a definition of *secure operating system* that we will use to assess several operating systems security approaches and specific implementations of those approaches. While no implementation completely satisfies this ideal definition, its use identifies the challenges in implementing operating systems that satisfy this ideal in practice. The aim is multi-fold. First, we want to understand the basic strengths of common security approaches. Second, we want to discover the challenges inherent to each of these approaches. These challenges often result in difficult choices in practical application. Third, we want to study the application of these approaches in practical environments to evaluate the effectiveness of these approaches to satisfy the ideal in practice. While it appears impractical to build an operating system that satisfies the ideal definition, we hope that studying these systems and their security approaches against the ideal will provide insights that enable the development of more effective security mechanisms in the future.

To return to the general definition of a secure operating system from the beginning of this section, we examine the general requirements of a secure operating system. To build any secure system requires that we consider how the system achieves its *security goals* under a set of threats (i.e., a *threat model*) and given a set of software, including the security mechanisms, that must be trusted [2] (i.e., a *trust model*).

## 1.2 SECURITY GOALS

A security goal defines the operations that can be executed by a system while still preventing unauthorized access. It should be defined at a high-level of abstraction, not unlike the way that an algorithm's worst-case complexity prescribes the set of implementations that satisfy that requirement. A security goal defines a requirement that the system's design can satisfy (e.g., the way pseudocode can be proven to fulfill the complexity requirement) and that a correct implementation must fulfill (e.g., the way that an implementation can be proven experimentally to observe the complexity).

[1] For example, the first description of criteria to verify that a system implements correct security mechanisms is called the Trusted Computer System Evaluation Criteria [304].

[2] We assume that hardware is trusted to behave as expected. Although the hardware devices may have bugs, the trust model that we will use throughout this book assumes that no such bugs are present.

Security goals describe how the system implements accesses to system resources that satisfy the following: *secrecy*, *integrity*, and *availability*. A system access is traditionally stated in terms of which *subjects* (e.g., processes and users) can perform which *operations* (e.g., read and write) on which *objects* (e.g., files and sockets). Secrecy requirements limit the objects that individual subjects can *read* because objects may contain secrets that not all subjects are permitted to know. Integrity requirements limit the objects that subjects can *write* because objects may contain information that other subjects *depend on* for their correct operation. Some subjects may not be trusted to modify those objects. Availability requirements limit the system resources (e.g., storage and CPU) that subjects may *consume* because they may exhaust these resources. Much of the focus in secure operating systems is on secrecy and integrity requirements, although availability may indirectly impact these goals as well.

The security community has identified a variety of different security goals. Some security goals are defined in terms of security requirements (i.e., secrecy and integrity), but others are defined in terms of function, in particular ways to limit function to improve security. An example of a goal defined in terms of security requirements is the *simple-security property* of the Bell-LaPadula model [23]. This goal states that a process cannot read an object whose secrecy classification is higher than the process's. This goal limits operations based on a security requirement, secrecy. An example of an functional security goal is the *principle of least privilege* [265], which limits a process to only the set of operations necessary for its execution. This goal is functional because it does not ensure that the secrecy and/or integrity of a system is enforced, but it encourages functional restrictions that may prevent some attacks. However, we cannot prove the absence of a vulnerability using functional security goals. We discuss this topic in detail in Chapter 5.

The task of the secure operating system developer is to define security goals for which the security of the system can be verified, so functional goals are insufficient. On the other hand, secrecy and integrity goals prevent function in favor of security, so they may be too restrictive for some production software. In the past, operating systems that enforced secrecy and integrity goals (i.e., the constrained systems above) were not widely used because they precluded the execution of too many applications (or simply lacked popular applications). Emerging technology, such as virtual machine technology (see Chapter 11), enables multiple, commercial software systems to be run in an isolated manner on the same hardware. Thus, software that used to be run on the same system can be run in separate, isolated virtual systems. It remains to be seen whether such isolation can be leveraged to improve system security effectively. Also, several general-purpose operating systems are now capable of expressing and enforcing security goals. Whether these general-purpose systems will be capable of implementing security goals or providing sufficient assurance for enforcing such goals is unclear. However, in either case, security goals must be defined and a practical approach for enforcing such goals, that enables the execution of most popular software in reasonable ways, must be identified.

## 1.3 TRUST MODEL

A system's *trust model* defines the set of software and data upon which the system depends for correct enforcement of system security goals. For an operating system, its trust model is synonymous with the system's *trusted computing base* (TCB).

Ideally, a system TCB should consist of the minimal amount of software necessary to enforce the security goals correctly. The software that must be trusted includes the software that defines the security goals and the software that enforces the security goals (i.e., the operating system's security mechanism). Further, software that bootstraps this software must also be trusted. Thus, an ideal TCB would consist of a bootstrapping mechanism that enables the security goals to be loaded and subsequently enforced for lifetime of the system.

In practice, a system TCB consists of a wide variety of software. Fundamentally, the enforcement mechanism is run within the operating system. As there are no protection boundaries between operating system functions (i.e., in the typical case of a monolithic operating system), the enforcement mechanism must trust all the operating system code, so it is part of the TCB.

Further, a variety of other software running outside the operating system must also be trusted. For example, the operating system depends on a variety of programs to authenticate the identity of users (e.g., `login` and `SSH`). Such programs must be trusted because correct enforcement of security goals depends on correct identification of users. Also, there are several services that the system must trust to ensure correct enforcement of security goals. For example, windowing systems, such as the X Window System [345], perform operations on behalf of all processes running on the operating system, and these systems provide mechanisms for sharing that may violate the system's security goals (e.g., cut-and-paste from one application to another) [85]. As a result, the X Window Systems and a variety of other software must be added to the system's TCB.

The secure operating system developer must prove that their systems have a viable trust model. This requires that: (1) the system TCB must mediate all security-sensitive operations; (2) verification of the correctness of the TCB software and its data; and (3) verification that the software's execution cannot be tampered by processes outside the TCB. First, identifying the TCB software itself is a nontrivial task for reasons discussed above. Second, verifying the correctness of TCB software is a complex task. For general-purpose systems, the amount of TCB software outside the operating system is greater than the operating system software that is impractical to verify formally. The level of trust in TCB software can vary from software that is formally-verified (partially), fully-tested, and reviewed to that which the user community trusts to perform its appointed tasks. While the former is greatly preferred, the latter is often the case. Third, the system must protect the TCB software and its data from modification by processes outside the TCB. That is, the integrity of the TCB must be protected from the threats to the system, described below. Otherwise, this software can be tampered, and is no longer trustworthy.

## 1.4 THREAT MODEL

A *threat model* defines a set of operations that an *attacker* may use to *compromise* a system. In this threat model, we assume a powerful attacker who is capable of injecting operations from the network and may be in control of some of the running software on the system (i.e., outside the trusted computing base). Further, we presume that the attacker is actively working to violate the system security goals. If an attacker is able to find a vulnerability in the system that provides access to secret information (i.e., violate secrecy goals) or permits the modification of information that subjects depend on (i.e., violate integrity goals), then the attacker is said to have compromised the system.

Since the attacker is actively working to violate the system security goals, we must assume that the attacker may try any and all operations that are permitted to the attacker. For example, if an attacker can only access the system via the network, then the attacker may try to send any operation to any processes that provide network access. Further, if an attacker is in control of a process running on the system, then the attacker will try any means available to that process to compromise system security goals.

This threat model exposes a fundamental weakness in commercial operating systems (e.g., UNIX and Windows); they assume that all software running on behalf of a subject is trusted by that subject. For example, a subject may run a word processor and an email client, and in commercial systems these processes are trusted to behave as the user would. However, in this threat model, both of these processes may actually be under the control of an attacker (e.g., via a document macro virus or via a malicious script or email attachment). Thus, a secure operating system cannot trust processes outside of the TCB to behave as expected. While this may seem obvious, commercial systems trust any user process to manage the access of that user's data (e.g., to change access rights to a user's files via `chmod` in a UNIX system). This can result in the leakage of that user's secrets and the modification of data that the user depends on.

The task of a secure operating system developer is to protect the TCB from the types of threats described above. Protecting the TCB ensures that the system security goals will always be enforced regardless of the behavior of user processes. Since user processes are untrusted, we cannot depend on them, but we can protect them from threats. For example, secure operating system can prevent a user process with access to secret data from leaking that data, by limiting the interactions of that process. However, protecting the TCB is more difficult because it interacts with a variety of untrusted processes. A secure operating system developer must identify such threats, assess their impact on system security, and provide effective countermeasures for such threats. For example, a trusted computing base component that processes network requests must identify where such untrusted requests are received from the network, determine how such threats can impact the component's behavior, and provide countermeasures, such as limiting the possible commands and inputs, to protect the component. The secure operating system developer must ensure that all the components of the trusted computing base prevent such threats correctly.

## 1.5 SUMMARY

While building a truly secure operating system may be infeasible, operating system security will improve immensely if security becomes a focus. To do so requires that operating systems be designed to enforce security goals, provide a clearly-identified trusted computing base that defines a trust model, define a threat model for the trusted computing base, and ensure protection of the trusted computing base under that model.