CHAPTER 4

# Security in Ordinary Operating Systems

In considering the requirements of a secure operating system, it is worth considering how far ordinary operating systems are from achieving these requirements. In this chapter, we examine the UNIX and Windows operating systems and show why they are fundamentally not secure operating systems. We first examine the history these systems, briefly describe their protection systems, then we show, using the requirements of a secure operating system defined in Chapter 2, why ordinary operating systems are inherently insecure. Finally, we examine common vulnerabilities in these systems to show the need for secure operating systems and the types of threats that they will have to overcome.

## 4.1 SYSTEM HISTORIES

### 4.1.1 UNIX HISTORY

UNIX is a multiuser operating system developed by Dennis Ritchie and Ken Thompson at AT&T Bell Labs [266]. UNIX started as a small project to build an operating system to play a game on an available PDP-7 computer. However, UNIX grew over the next 10 to 15 years into a system with considerable mindshare, such that a variety of commercial UNIX efforts were launched. The lack of coherence in these efforts may have limited the market penetration of UNIX, but many vendors, even Microsoft, had their own versions. UNIX remains a significant operating system today, embodied in many systems, such as Linux, Sun Solaris, IBM AIX, the various BSD systems, etc.

Recall from Chapter 3 that Bell Labs was a member of the Multics consortium. However, Bell Labs dropped out of the Multics project in 1969, primarily due to delays in the project. Ken Thompson adapted some of the ideas of Multics when he initiated the construction of a system that was named as a pun on the Multics system, UNICS (UNIplexed Information and Computing Service). Eventually and mysteriously, the system was renamed UNIX, but the project had begun.

UNIX gained mindshare for a number of reasons. Ritchie rewrote UNIX in his new programming language C which enabled UNIX to be the first portable operating system. This enabled the development of a UNIX community, since lots of people could run UNIX on a variety of different hardware. Next, an application program interface was designed for UNIX which enabled programmers to write application easily, without resorting to assembly language, and these applications ran across the variety of UNIX-supported platforms. Finally, UNIX was truly simplified when compared to Multics. While UNIX adopted many Multics principles, such as hierarchical file systems, virtual memory, and encrypted passwords, UNIX was far simpler. UNIX aimed for a small base program called the *kernel* with a standard interface to simplify the development of applications. As a result,

the code size of UNIX (at the time) was smaller than Multics, UNIX performed better, and UNIX was easier to program and administer.

As a streamlined descendant of Multics, UNIX adopted several of the Multics security features, such as password storage, protection ring usage, access control lists, etc., but most were streamlined as well. Since UNIX was not a government-funded project like Multics, it was built with different security goals in mind. For UNIX, the goal was to develop a common platform (e.g., devices and file system) that could be shared by several users. As a result, the security problem became one of *protection* [1], where the goal is to protect the users' data from inadvertent errors in their programs. However, protection does not ensure that secrecy and integrity goals (i.e., security) can be achieved (see Chapter 5). Security enforcement requires that a system's security mechanisms can enforce system security goals even when all the software outside the trusted computing base is malicious. Thus, when UNIX systems were connected to untrusted users via the Internet, a variety of design decisions made for protection no longer applied. As we will discuss, the ordinary UNIX security mechanisms are not capable of enforcing the requirements of a secure operating system. A variety of efforts have aimed to extend or replace the insecure mechanisms for ordinary UNIX systems with mechanisms that may achieve the requirements of a secure operating system (see Chapter 2), as we describe in Chapters 7 and 9.

### 4.1.2 WINDOWS HISTORY

The history of the Microsoft Windows operating system goes back to the introduction of MS-DOS, which was the original operating system for IBM personal computers introduced in 1981 [24]. MS-DOS was constructed from the Quick and Dirty Operating System (QDOS) built by Tim Paterson that Microsoft purchased from his employer Seattle Computer Products. QDOS was itself based on an early microcomputer operating system called the Control Program for Microcomputers (CP/M) [68, 75]. Compared to other operating systems of the time, such as Multics and UNIX, MS-DOS was a very limited system. It was not a true multitasking system, and did not use many of the features of the x86 processor. Over the next 20 years, Microsoft made improvements to MS-DOS to support more efficient and flexible use of the x86 hardware.

Windows was originally a GUI for MS-DOS, but its visibility soon led to using its name for the subsequent operating systems that Microsoft released. Early Windows systems were based on various versions of MS-DOS, but MS-DOS became less fundamental to the later "Windows 9x" systems. A second, independent line of systems based on the NT kernel emerged starting with the Windows NT 4.0. In 2000, the Windows systems derived from the original MS-DOS codebase were discontinued. At this point, the Windows brand of operating systems dominated the desktop computing market and spanned most computing devices, but the lack of focus on security in Windows operating systems was becoming a significant limitation in these systems.

As the initial focus of the Windows operating system was on microcomputer platforms envisioned for a single user and disconnected from any network, security was not a feature of such

[1] Named after the protection system in Lampson's famous paper [176] which achieves the same security goal.

systems. Users administered their systems, uploading new programs as they were purchased. However, the emergence of the world-wide web made connecting Windows computers to the network fundamental to its use, and the networked services that users leveraged, such as email, web clients, easy program download, etc., introduced vulnerabilities that the Windows systems were not designed to counter. The usability model of Windows as a open, flexible, user-administered platform, plus its ubiquity, made it an easy target for attackers. Further, Microsoft was slow to address such threats. In 2000, features were nearly always enabled by default, leading to world-wide compromises due to Windows vulnerabilities (e.g., Code Red and variants [88, 334]). Microsoft has focused with some success on reducing its vulnerabilities through better code development practices [139], code analysis tools [210], and more secure configuration settings. However, improvements in the security features of the Windows operating systems have been less effective. The Windows 2000-based access control system is complex and largely unused [303], the Windows operating system trusted computing base is extremely large (50 million lines of source code in the operating system alone), and recent security enhancements for Windows Vista [152] are both insufficient to provide complete integrity protection [221, 220] and so invasive as to be unpopular [243].

## 4.2 UNIX SECURITY

We provide a brief outline of a UNIX system prior to examining the security details. Those interested in a comprehensive description of UNIX system concepts are encouraged to read one of the many books on the subject [119, 201, 192].

A running UNIX system consists of an *operating system kernel* and many *processes* each running a program. A protection ring boundary isolates the UNIX kernel from the processes. Each process has its own *address space*, that defines the memory addresses that it can access. Modern UNIX systems define address spaces primarily in terms of the set of *memory pages* that they can access [2]. UNIX uses the concept of a *file* for all persistent system objects, such as secondary storage, I/O devices, network, and interprocess communication. A UNIX process is associated with an *identity*, based on the user associated with the process, and access to files is limited by the process's identity.

UNIX security aims to protect users from each other and the system's trusted computing base (TCB) from all users. Informally, the UNIX TCB consists of the kernel and several processes that run with the identity of the privileged user, `root` or superuser. These `root` processes provide a variety of services, including system boot, user authentication, administration, network services, etc. Both the kernel and `root` processes have full system access. All other processes have limited access based on their associated user's identity.

### 4.2.1 UNIX PROTECTION SYSTEM

UNIX implements a classical protection system (see Definition 2.1 in Chapter 2), not the secure protection system (see Definition 2.4). As stated in Definition 2.1, a UNIX protection system

[2] Segmentation is still supported in most modern processors, but it is not used as the primary access boundary in UNIX systems anymore, as it was in Multics.

consists of a protection state and a set of operations that enable processes to modify that state. Thus, UNIX is a *discretionary access control* (DAC) system. However, UNIX does have some aspects of the secure protection system in Definition 2.4. First, the UNIX protection system defines a *transition state* that describes how processes change between protection domains. Second, the *labeling state* is largely ad hoc. Trusted services associate processes with user identities, but users can control the assignment of permissions to system resources (i.e., files). In the final analysis, these mechanisms and the discretionary protection system are insufficient to build a system that satisfies the secure operating system requirements (see Definition 2.6 in Chapter 2).

Recall that a protection state describes the operations that the system's subjects can perform on that system's objects. The UNIX protection state associates process identities (subjects) with their access to files (objects). Each UNIX process identity consists of a *user id* (UID), a *group id* (GID), and a set of *supplementary groups*. These are used in combination to determine access as described below [3].

All UNIX resources are represented as files. The protection state specifies that subjects may perform read, write, and execute operations on files, with the standard meaning of these operations. While directories are not files, they are represented as files in the UNIX protection state, although the operations have different semantics (e.g., `execute` means search for a directory).

Files are also associated with an owner UID and an owner GID which conveys special privileges to processes with these identities. A process with the owner UID can modify any aspect of the protection state for this file. Processes with either the owner UID and group GID may obtain additional rights to access the file as described below.

The limited set of objects and operations enabled UNIX designers to use a compressed access control list format called *UNIX mode bits*, to specify the access rights of identities to files. Mode bits define the rights of three types of subjects: (1) the file owner UID; (2) the file group GID; and (3) all other subjects. Using mode bits authorization is performed as follows. First, the UNIX authorization mechanism checks whether the process identity's UID corresponds to the owner UID of the file, and if so, uses the mode bits for the owner to authorize access. If the process identity's GID or supplementary groups correspond to the file's group GID, then the mode bits for the group permissions are used. Otherwise, the permissions assigned to all others are used.

**Example 4.1.** UNIX mode bits are of the form {owner bits, group bits, others bits} where each element in the tuple consists of a read bit, a write bit, and an execute bit. The mode bits:

```
rwxr--r--
```

mean that a process with the same UID as the owner can read, write, or execute the file, a process with a GID or supplementary group that corresponds to the file's group can read the file, and others can also only read the file.

[3] A process's user identity is actually represented by a set of UIDs for effective, real, and file system access. These details are important to preventing vulnerabilities, see Section 4.2.4, but for clarity we defer their definition until that section.

Suppose a set of files have the following owners, groups, and others mode bits as described below:

```
Name  Owner     Group      Mode Bits
foo   alice     faculty    rwxr--r--
bar   bob       students   rw-rw-r--
baz   charlie   faculty    rwxrwxrwx
```

Then, processes running as `alice` with the group `faculty` can read, write, or execute `foo` and `baz`, but only read `bar`. For `bar`, Alice does not match the UID (bob), nor have the associated group (students). The process has the appropriate owner to gain all privileges for `foo` and the appropriate group to gain privileges to `baz`.

As described above, the UNIX protection system is a discretionary access control system. Specifically, this means that a file's mode bits, owner UID, or group GID may be changed by any UNIX processes run by the file's owner (i.e., that have the same UID as the file owner). If we trust all user processes to act in the best interests of the user, then the user's security goals can be enforced. However, this is no longer a reasonable assumption. Nowadays, users run a variety of processes, some of which may be supplied by attackers and others may be vulnerable to compromise from attackers, so the user will have no guarantee that these processes will behave consistently with the user's security goals. As a result, a secure operating system cannot use discretionary access control to enforce user security goals.

Since discretionary access control permits users to change their files owner UID and group GID in addition to the mode bits, file labeling is also discretionary. A secure protection system requires a mandatory *labeling state*, so this is another reason that UNIX systems cannot satisfy the requirements of a secure operating system.

UNIX processes are labeled by trusted services from a set of labels (i.e., user UIDs and group GIDs) defined by trusted administrators, and child processes inherit their process identity from their parent. This mandatory approach to labeling processes with identities would satisfy the secure protection system requirements, although it is rather inflexible.

Finally, UNIX mode bits also include a specification for protection domain transitions, called the `setuid` bit. When this bit is set on a file, any process that executes the file with automatically perform a protection domain transition to the file's owner UID and group GID. For example, if a `root` process sets the `setuid` bit on a file that it owns, then any process that executes that file will run under the `root` UID. Since the `setuid` bit is a mode bit, it can be set by the file's owner, so it is also managed in a discretionary manner. A secure protection state requires a mandatory *transition state* describe all protection domain transitions, so the use of discretionary `setuid` bits is insufficient.

### 4.2.2 UNIX AUTHORIZATION

The UNIX authorization mechanism controls each process's access to files and implements protection domain transitions that enable a process to change its identity. The authorization mechanism runs

in the kernel, but it depends on system and user processes for determining its authorization queries and its protection state. For these and other reasons described in the UNIX security analysis, the UNIX authorization mechanism does not implement a reference monitor. We prove this in the Section 4.2.3 below.

UNIX authorization occurs when files are opened, and the operations allowed on the file are verified on each file access. The requesting process provides the name of the file and the operations that will be requested upon the file in the `open` system call. If authorized, UNIX creates a *file descriptor* that represents the process's authorized access to perform future operations on the file. File descriptors are stored in the kernel, and only an index is returned to the process. Thus, file descriptors are a form of *capability* (see Chapter 2 for the definition and Chapter 10 for a discussion on capability-based systems). User processes present their file descriptor index to the kernel when they request operations on the files that they have opened.

UNIX authorization controls traditional file operations by mediating file `open` for read, write, and execute permissions. However, the use of these permissions does not always have the expected effect: (1) these permissions and their semantics do not always enable adequate control and (2) some objects are not represented as files, so they are unmediated. If a user has read access to a file, this is sufficient to perform a wide-variety of operations on the file besides reading. For example, simply via possession of a file descriptor, a user process can perform any ad hoc command on the file using the system calls `ioctl` or `fcntl`, as well as read and modify file metadata. Further, UNIX does not mediate all security-sensitive objects, such as network communications. Host firewalls provide some control of network communication, but they do not restrict network communication by process identity.

The UNIX authorization mechanism depends on user-level authentication services, such as `login` and `sshd`, to determine the process identity (i.e., UID, GID, and supplementary groups, see Section 4.2.1). When a user logs in to a system, her processes are assigned her login identity. All subsequent processes created in this login session inherit this identity unless there is a domain transition (see below). Such user-level services aleelo

such as `login` or `sshd`, can change the identity of a process. For example, when a user logs in, the login program must change the process identity of the user's first process, her shell, to the user to ensure correct access control. In the second case, the use of the `setuid` bit on a file is typically used to permit a lower privileged entity to execute a higher privileged program, almost always as `root`. For example, when a user wishes to change her password, she uses the `passwd` program. Since the `passwd` program modifies the password file, it must be privileged, so a process running with the user's identity could not change the password file. The `setuid` bit on the `root`-owned, `passwd` executable's file is set, so when any user executes `passwd`, the resultant process identity transitions to `root`. While the identity transition does not impact the user's other processes, the writers of the `passwd` program must be careful not to allow the program to be tricked into allowing the user to control how `passwd` uses its additional privileges.

UNIX also has a couple of mechanisms that enable a user to run a process with a reduced set of permissions. Unfortunately, these mechanisms are difficult to use correctly, are only available to `root` processes, and can only implement modest restrictions. First, UNIX systems have a special principal `nobody` that owns no files and belongs to no groups. Therefore, a process's permissions can be restricted by running as `nobody` since it never has owner or group privileges. Unfortunately, `nobody`, like all subjects, has `others` privileges. Also, since only `root` can do a `setuid` only a superuser process can change the process identity to `nobody`. Second, UNIX `chroot` can be used to limit a process to a subtree of the file system [262]. Thus, the process is limited to only its rights to files within that subtree. Unfortunately, a `chroot` environment must be setup carefully to prevent the process from escaping the limited domain. For example, if an attacker can create `/etc/passwd` and `/etc/shadow` files in the subtree, she can add an entry for `root`, login as this root, and escape the `chroot` environment (e.g., using `root` access to kernel memory). Also, a `chroot` environment can only be setup by a `root` process, so it is not usable to regular system users. In practice, neither of these approaches has proven to be an effective way to limit process permissions.

### 4.2.3 UNIX SECURITY ANALYSIS

If UNIX can be a secure operating system, it must satisfy the secure operating system requirements of Chapter 2. However, UNIX fails to meet any of these requirements.

1. **Complete Mediation**: How does the reference monitor interface ensure that all security-sensitive operations are mediated correctly?

   The UNIX reference monitor interface consists of hooks to check access for file or inode permission on some system calls. The UNIX reference monitor interface authorizes access to the objects that the kernel will use in its operations.

   A problem is that the limited set of UNIX operations (read, write, and execute) is not expressive enough to control access to information. As we discussed in Section 4.2.2, UNIX permits modifications to files without the need for write permission (e.g., `fcntl`).

2. **Complete Mediation**: Does the reference monitor interface mediate security-sensitive operations on all system resources?

   UNIX authorization does not provide complete mediation of all system resources. For some objects, such as network communications, UNIX itself provides no authorization at all.

3. **Complete Mediation**: How do we verify that the reference monitor interface provides complete mediation?

   Since the UNIX reference monitor interface is placed where the security-sensitive operations are performed, it difficult to know whether all operations have been identified and all paths have been mediated. No specific approach has been used to verify complete mediation.

4. **Tamperproof**: How does the system protect the reference monitor, including its protection system, from modification?

   The reference monitor and protection system are stored in the kernel, but this does not guarantee tamper-protection. First, the protection system is discretionary, so it may be tampered by any running process. Untrusted user processes can modify permissions to their user's data arbitrarily, so enforcing security goals on user data is not possible.

   Second, the UNIX kernel is not as protected from untrusted user processes as the Multics kernel is. Both use protection rings for isolation, but the Multics system also explicitly specifies *gates* for verifying the legality of the ring transition arguments. While UNIX kernels often provide procedures to verify system call arguments, such procedures are may be misplaced.

   Finally, user-level processes have a variety of interfaces to access and modify the kernel itself above and beyond system calls, ranging from the ability to install kernel modules to special file systems (e.g., `/proc` or *sysfs*) to interfaces through `netlink` sockets to direct access to kernel memory (e.g., via the device file`/dev/kmem`). Ensuring that these interfaces can only be accessed by trusted code has become impractical.

5. **Tamperproof**: Does the system's protection system protect the trusted computing base programs?

   In addition to the kernel, the UNIX TCB consists of *all* `root` processes, including all processes run by a user logged in as a `root` user. Since these processes could run any program, guaranteeing the tamper-protection of the TCB is not possible. Even ignoring `root` users, the amount of TCB code is far too large and faces far too many threats to claim a tamperproof trusting computing base. For example, several `root` processes have open network ports that may be used as avenues to compromise these processes. If any of these processes is compromised, the UNIX system is effectively compromised as there is no effective protection among `root` processes.

Also, any `root` process can modify any aspect of the protection system. As we show below, UNIX `root` processes may not be sufficiently trusted or protected, so unauthorized modification of the protection system, in general, is possible. As a result, we cannot depend on a tamperproof protection system in a UNIX system.

6. **Verifiable**: What is basis for the correctness of the system's TCB?

   Any basis for correctness in a UNIX system is informal. The effectively unbounded size of the TCB prevents any effective formal verification. Further, the size and extensible nature of the kernel (e.g., via new device drivers and other kernel modules) makes it impractical to verify its correctness.

7. **Verifiable**: Does the protection system enforce the system's security goals?

   Verifiability enforcement of security goals is not possible because of the lack of complete mediation and the lack of tamperproofing. Since we cannot express a policy rich enough to prevent unauthorized data leakage or modification, we cannot enforce secrecy or integrity security goals. Since we cannot prove that the TCB is protected from attackers, we cannot prove that the system will be remain able to enforce our intended security goals, even if they could be expressed properly.

### 4.2.4 UNIX VULNERABILITIES

A secure operating system must protect its trusted computing base from compromise in order to implement the reference monitor guarantees as well. In this section, we list some of the vulnerabilities that have been found in UNIX systems over the years that have resulted in the compromise of the UNIX trusted computing base. This list is by no means comprehensive. Rather, we aim to provide some examples of the types of problems encountered when the system design does not focus on protecting the integrity of the trusted computing base.

**Network-facing Daemons** UNIX has several `root` (i.e., TCB) processes that maintain network ports that are open to all remote parties (e.g., `sshd`, `ftpd`, `sendmail`, etc.), called *network-facing daemons*. In order to maintain the integrity of the system's trusted computing base, and hence achieve the reference monitor guarantees, such process must protect themselves from such input. However, several vulnerabilities have been reported for such processes, particularly due to buffer overflows [232, 318], enabling remote attackers to compromise the system TCB. Some of these daemons have been redesigned to remove many of such vulnerabilities (e.g., Postfix [317, 73] as a replacement for `sendmail` and privilege-separated SSH [251]), but a comprehensive justification of integrity protection for the resulting daemons is not provided. Thus, integrity protection of network-facing dameons in UNIX is incomplete and ad hoc.

Further, some network-facing daemons, such as remote login daemons (e.g., `telnet`, `rlogin`, etc.) `ftpd`, and NFS, puts an undo amount of trust in the network. The remote login daemons and

`ftpd` are notorious for sending passwords in the clear. Fortunately, such daemons have been obsoleted or replaced by more secure versions (e.g., `vsftpd` for `ftpd`). Also, NFS is notorious for accepting any response to a remote file system request as being from a legitimate server [38]. Network-facing daemons must additionally protect the integrity of their secrets and authenticate the sources of remote data whose integrity is crucial to the process.

**Rootkits** Modern UNIX systems support extension via kernel modules that may be loaded dynamically into the kernel. However, a malicious or buggy module may enable an attacker to execute code in the kernel, with full system privileges. A variety of malware packages, called *rootkits*, have been created for taking advantage of kernel module loading or other interfaces to the kernel available to `root` processes. Such rootkits enable the implementation of attacker function and provide measures to evade from detection. Despite efforts to detect malware in the kernel [244, 245], such rootkits are difficult to detect, in general, [17].

**Environment Variables** UNIX systems support *environment variables*, system variables that are available to processes to convey state across applications. One such variable is `LIBPATH` whose value determines the search order for dynamic libraries. A common vulnerability is that an attacker can change `LIBPATH` to load an attacker-provided file as a dynamic library. Since environment variables are inherited when a child process is created, an untrusted process can invoke a TCB program (e.g., a program file which `setuid`'s to `root` on invocation, see Section 4.2.2) under an untrusted environment. If the TCB process depends on dynamic libraries and does not set the `LIBPATH` itself, it may be vulnerable to running malicious code. As many TCB programs can be invoked via `setuid`, this is a widespread issue.

Further, TCB programs may be vulnerable to any input value supplied by an untrusted process, such as malicious input arguments. For example, a variety of program permit the caller to define the configuration file of the process. A configuration file typically describes all the other places that the program should look for inputs to describe how it should function, sometimes including the location of libraries that it should use and the location of hosts that provide network information. If the attack can control the choice of a program's configuration file, she often has a variety of ways to compromise the running process. Any TCB program must ensure their integrity regardless of how they are invoked.

**Shared Resources** If TCB processes share resources with untrusted processes, then they may be vulnerable to attack. A common problem is the sharing of the `/tmp` directory. Since any process can create files in this directory, an untrusted process is able to create files in this directory and grant other processes, in particular a TCB process, access to such files as well. If the untrusted process can guess the name of TCB process's `/tmp` file, it can create this file in advance, grant access to the TCB process, and then have access itself to a TCB file. TCB processes can prevent this problem

by checking for the existence of such files upon creation (e.g., using the `O_CREAT` flag). However, programmers have been prone to forget such safeguards. TCB process must take care when using any objects shared by untrusted processes.

**Time-of-Check-to-Time-of-Use (TOCTTOU)** Finally, UNIX has been prone to a variety of attacks where untrusted processes may change the state of the system between the time an operation is authorized and the time that the operation is performed. If such a change enables an untrusted process to access a file that would not have been authorized for, then this presents a vulnerability. The attack was first identified by Dilger and Bishop [30] who gave it the moniker *time-of-check-to-time-of-use attacks* or TOCTTOU attacks. In the classical example, a `root` process uses the system call `access` to determine if the user for whom the process is running (e.g., the process was initiated by a `setuid`) has access to a particular file `/tmp/X`. However, after the `access` system call authorizes the file access and before the file `open`, the user may change the binding between the file name and the actual file object (i.e., `inode`) accessed. This can be done by change the file `/tmp/X` to a symbolic link to the target file `/etc/shadow`. As a result, UNIX added a flag, so the `open` request could prevent traversal via symbolic links. However, the UNIX file system remains susceptible to TOCTTOU attacks because the mapping between file names and actual file objects (`inodes`) can be manipulated by the untrusted processes.

As a result of the discretionary protection system, the size of the system TCB, and these types of vulnerabilities, converting a UNIX system to a secure operating system is a significant challenge. Ensuring that TCB processes protect themselves, and thus protect a reference monitor from tampering, is a complex undertaking as untrusted processes can control how TCB processes are invoked and provide inputs in multiple ways: network, environment, and arguments. Further, untrusted processes may use system interfaces to manipulate any shared resources and may even change the binding between object name and the actual object. We will discuss the types of changes necessary to convert an ordinary UNIX system to a system that aims to satisfy the secure operating system definition in Chapters 7 and 9, so we will see that several fundamental changes are necessary to overcome these problems. Even then, the complexity of UNIX systems and their trusted computing base makes satisfying the tamperproof and verifiability requirements of the reference monitor concept very difficult.

## 4.3 WINDOWS SECURITY

In this section, we will show that Windows operating systems also fail to meet the requirements of a secure operating system. This section will be much briefer than the previous examination of UNIX as many of the concepts are similar. For example, Windows also supports processes with their own address spaces that are managed by a ring-protected kernel. For a detailed description of the Windows access control system examined in this section, circa Windows 2000, see Swift et al. [303].

### 4.3.1 WINDOWS PROTECTION SYSTEM

The Windows 2000 protection system [5], like the UNIX protection system, provides a discretionary access control model for managing protection state, object labeling, and protection domain transitions. The two protection systems manly differ in terms of flexibility (e.g., the Windows system is extensible) and expressive power (e.g., the Windows system enables the description of a wider variety of policies). Unfortunately, when we compare the Windows protection system to the definition of a secure protection system, we find that improvements in flexibility and expressive power actually make the system more difficult to secure.

Specifically, the Windows protection system differs from UNIX mainly in the variety of its objects and operations and the additional flexibility it provides for assigning them to subjects. When the Windows 2000 access control model was being developed, there were a variety of security systems being developed that provided administrators with extensible policy languages that permitted flexible policy specification, such as the Java 2 model [117]. While these models address some of the shortcomings of the UNIX model by enabling the expression of any protection state, they do not ensure a secure system.

Subjects in Windows are similar to subjects in UNIX. In Windows, each process is assigned a *token* that describes the process's identity. A process identity consists of user security identifier (principal SID, analogous to a UNIX UID), a set of group SIDs (rather than a single UNIX GID and a set of supplementary groups), a set of alias SIDs (to enable actions on behalf of another identity), and a set of privileges (ad hoc privileges just associated with this token). A Windows identity is still associated with a single user identity, but a process token for that user may contain any combination of rights.

Unlike UNIX, Windows objects can belong to a number of different data types besides files. In fact, applications may define new data types, and add them to the *active directory*, the hierarchical name space for all objects known to the system. From an access control perspective, object types are defined by their set of operations. The Windows model also supports a more general view of the operations that an object type may possess. Windows defines up to 30 operations per object type, including some operations that are specific to the data type [74]. This contrasts markedly with the read, write, and execute operations in the UNIX protection state. Even for file objects, the Windows protection system defines many more operations, such as operations to access file attributes and synchronize file operations. In addition, application may add new object types and define their own operations.

The other major difference between a Windows and UNIX protection state is that Windows supports arbitrary access control lists (ACLs) rather than the limited mode bits approach of UNIX. A Windows ACL stores a set of access control entries (ACEs) that describe which operations an SID (user, group, or alias) can perform on that object [6]. The operations in an ACE are interpreted based on the object type of the target object. In Windows, ACEs may either grant or deny an operation.

[5] We simply refer to this as the Windows protection system for the rest of the chapter.
[6] Remember that access control lists are stored with the object, and state which subjects can access that object.

Thus, Windows uses negative access rights, whereas UNIX does not, generating some differences in their authorization mechanisms.

**Example 4.2.** Figure 4.1 shows an example ACL for an object `foo`. `foo`'s ACL contains three ACEs. The field *principal SID* specifies the SID to which the ACE applies. These ACE apply to

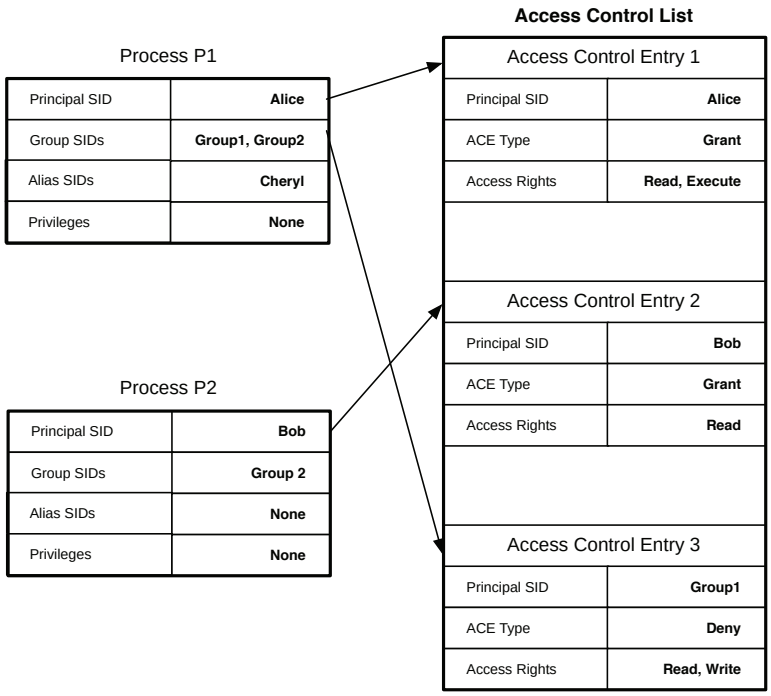


**Figure 4.1:** Windows Access Control Lists (ACLs) and process tokens for Examples 4.2 and 4.3

the SIDs `Alice`, `Bob`, and `Group1`. The other two important fields in an ACE are its *type* (grant or deny) and the *access rights* (a bitmask). The `Alice` and `Bob` ACEs grant rights, and the `Group1` ACE denies access to certain rights. The access rights bitmask is interpreted based on the *object type* field in the ACE. We describe how the ACL is used in authorization in the next section.

### 4.3.2 WINDOWS AUTHORIZATION

Windows authorization queries are processed by a specific component called the *Security Reference Monitor* (SRM). The SRM is a kernel component that takes a process token, an object SID, and a set of operations, and it returns a boolean result of an authorization query. The SRM uses the object SID to retrieve its ACL from which it determines the query result.

Because of the negative permissions, the way that the SRM processes authorization queries is more complicated than in the UNIX case. The main difference is that the ACEs in an ACL are ordered, and the ACEs are examined in that order. The SRM searches the ACEs until it finds a set of ACEs that permits the operation or a single ACE that denies the operation. If an ACE grants the necessary operations [7], then the request is authorized. However, if a deny ACE is encountered that includes one of the requested operations, then the entire request is denied.

**Example 4.3.** Returning to Example 4.2 above, the ACEs of the object's ACL are ordered as shown in Figure 4.1. Note that the ACE field for access rights is really a bitmap, but we list the operations to simplify understanding. Further, we specify the process tokens for two processes, `P1` and `P2`. Below, we show the authorization results for a set of queries by these processes for the target object.

```
P1, read: ok
P1, read, write: no
P2: read: ok
P2: read, write: no
```

Both P1 and P2 can read the target object, but neither can write the object. P1 cannot write the object because the P1 token include `Group1` which matches the deny ACE for writing. P2 cannot write the object because the ACE for `Bob` does not permit writing.

Mediation in Windows is determined by a set of object managers. Rather than a monolithic set of system calls to access homogeneous objects (i.e., files) in UNIX, each object type in Windows has an object manager that implements the functions of that type. While the Windows object managers all run in the kernel, the object managers are independent entities. This can be advantageous from a modularity perspective, but the fact that object managers may extend the system presents some challenges for mediation. We need to know that each new object manager mediates all operations and determines the rights for those operations correctly. There is no process for ensuring this in Windows.

In Windows, the trusted computing base consists of all system services and processing running as a trusted user identity, such as `Administrator` [8]. Windows provides a `setuid`-like mechanism for invoking Windows *Services* that run at a predefined privilege, at least sufficient to support all clients. Thus, vulnerabilities in such services would lead to system compromise. Further, the ease of software installation and complexity of the discretionary Windows access control model often result in users running as `Administrator`. In this case, any user program would be able to take control of the system. This is often a problem on Windows systems. With the release of Windows Vista, the Windows model is extended to prevent programs downloaded from the Internet from

[7] It may take multiple ACEs to grant all the requested operations, so this refers to the ACE that grants whatever remaining operations were requested.

[8] In addition, these services and processes may further depend on non-Administrator processes, which would make the system TCB even less secure.

automatically being able to write Windows applications and the Windows system, regardless of the user's process identity [152]. While this does provide some integrity protection, it does not fully protect the system's integrity. It prevents low integrity processes from writing to high integrity files, but does not prevent invocation, malicious requests, or spoofing the high integrity code into using a low integrity file. See Chapter 5 for the integrity requirements of a secure operating system.

Windows also provides a means for restricting the permissions available to a process flexibly, called *restricted contexts*. By defining a restricted context for a process, the permissions necessary to perform an operation must be available to both the process using its token and to the restricted context. That is, the permissions of a process running in a restricted context are the *intersection* of the restricted context and the process's normal permissions. Since a restricted context may be assigned an arbitrary set of permissions, this mechanism is much more flexible than the UNIX option of running as `nobody`. Also, since restricted contexts are built into the access control system, it less error-prone than and `chroot`. Nonetheless, restricted contexts are difficult for administrators to define correctly, so they are not used commonly, and not at all by the user community.

### 4.3.3 WINDOWS SECURITY ANALYSIS

Despite the additional expressive power offered by the Windows access control model, it also does not satisfy any of the reference monitor guarantees either. Although Windows can express any combination of permissions, it becomes more difficult to administer. In my informal polls, no users use the Windows permission model at all, whereas most at least were aware of how to use the UNIX model (although not always correctly). Windows is effectively no more or less secure than ordinary UNIX—they are both insecure.

1. **Complete Mediation**: How does the reference monitor interface ensure that all security-sensitive operations are mediated correctly?

   In Windows, mediation is provided by object managers. Without the source code, it is difficult to know where mediation is performed, but we would presume that object managers would authorize the actual objects used in the security-sensitive operations, similarly to UNIX.

2. **Complete Mediation**: Does the reference monitor interface mediate security-sensitive operations on all system resources?

   Object managers provide an opportunity for complete mediation, but provide no guarantee of mediation. Further, the set of managers may be extended, resulting in the addition of potentially insecure object managers. Without a formal approach that defines what each manager does and how it is to be secured, it will not be possible to provide a guarantee of complete mediation.

3. **Complete Mediation**: How do we verify that the reference monitor interface provides complete mediation?

   As for UNIX, no specific approach has been used to verify complete mediation.

4. **Tamperproof**: How does the system protect the reference monitor, including its protection system, for modification?

   Windows suffers from the same problems as UNIX when it comes to tampering. First, the protection system is discretionary, so it may be tampered by any running process. Untrusted user processes can modify permissions to their user's data arbitrarily, so enforcing security goals on user data is not possible. Since users have often run as `Administrator` to enable ease of system administration, any aspect of the protection system may be modified.

   Second, there are limited protections for the kernel itself. Like UNIX, a Windows kernel can be modified through kernel modules. In Microsoft Vista, a code signing process can be used to determine the certifier of a kernel module (i.e., the signer, not necessarily the writer of the module). Of course, the administrator (typically an end user) must be able to determine the trustworthiness of the signer. Security procedures that depend on the decision-making of users are often prone to failure, as users are often ignorant of the security implications of such decisions. Also, like UNIX, the Windows kernel also does not define protections for system calls (e.g., Multics *gates*).

5. **Tamperproof**: Does the system's protection system protect the trusted computing base programs?

   The TCB of Windows system is no better than that of UNIX. Nearly any program may be part of the Windows TCB, and any process running these programs can modify other TCB programs invalidating the TCB.

   Like UNIX, any compromised TCB process can modify the protection system invalidating the enforcement of system security goals, and modify the Windows kernel itself through the variety of interfaces provided to TCB processes to access kernel state.

   Unlike UNIX, Windows provides APIs to tamper with other processes in ways that UNIX does not. For example, Windows provides the `CreateRemoteThread` function, which enables a process to initiate a thread in another process [207]. Windows also provides functions for writing a processes memory via `OpenProcess` and `WriteProcessMemory`, so one process can also write the desired code into that process prior to initiating a thread in that process. While all of these operations require the necessary access rights to the other process, usually requiring a change in privileges necessary for debugging a process (via the `AdjustTokenPrivileges`). While such privileges are typically only available to processes under the same SID, we must verify that these privileges cannot be misused in order to ensure tamper-protection of our TCB.

6. **Verifiable**: What is basis for the correctness of the system's trusted computing base?

   As for UNIX, any basis for correctness is informal. Windows also has an unbounded TCB and extensible kernel system that prevent any effective formal verification.

7. **Verifiable**: Does the protection system enforce the system's security goals?

   The general Windows model enables any permission combination to be specified, but no particular security goals are defined in the system. Thus, it is not possible to tell whether a system is secure. Since the model is more complex than the UNIX model and can be extended arbitrarily, this makes verifying security even more difficult.

### 4.3.4 WINDOWS VULNERABILITIES

Not surprisingly given its common limitations, Windows suffers from the same kinds of vulnerabilities as the UNIX system (see Section 4.2.4). For example, there are books devoted to constructing Windows rootkits [137]. Here we highlight a few vulnerabilities that are specific to Windows systems or are more profound in Windows systems.

**The Windows Registry** The Windows Registry is a global, hierarchical database to store data for all programs [206]. When a new application is loaded it may update the registry with application-specific, such as security-sensitive information such as the paths to libraries and executables to be loaded for the application. While each registry entry can be associated with a security context that limits access, such limitations are generally not effectively used. For example, the standard configuration of *AOL* adds a registry entry that specifies the name of a Windows library file (i.e., DLL) to be loaded with AOL software [120]. However, the permissions were set such that any user could write the entry.

This use of the registry is not uncommon, as vendors have to ensure that their software will execute when it is downloaded. Naturally, a user will be upset if she downloads some newly-purchased software, and it does not execute correctly because it could not access its necessary libraries. Since the application vendors cannot know the ad hoc ways that a Windows system is administered, they must turn on permissions to ensure that whatever the user does the software runs. If the registry entry is later used by an attacker to compromise the Windows system, that is not really the application vendor's problem—selling applications is.

**Administrator Users** We mentioned in the Windows security evaluation that traditionally users ran under the identity `Administrator` or at least with administrative privileges enabled. The reason for this is similar to the reason that broad access is granted to registry entries: the user also wants to be sure that they can use what function is necessary to enable the system to run. If the user downloads some computer game, the user would need special privileges to install the game, and likely need special privileges to run the device-intensive game program. The last thing the user wants is to have to figure out why the game will not run, so enabling all privileges works around this issue.

UNIX systems are generally used by more experienced computer users who understand the difference between installing software (e.g., run `sudo`) and the normal operation of the computer. As

a result, the distinction between `root` users and `sudo` operations has been utilized more effectively in UNIX.

**Enabled By Default** Like users and software vendors, Windows deployments also came with full permissions and functionality enabled. This resulted in the famous Code Red worms [88] which attacked the SQL server component of the Microsoft IIS web server. Many people who ran IIS did not have an SQL server running or even knew that the SQL server was enabled by default in their IIS system. But in these halcyon times, IIS web servers ran with all software enabled, so attackers could send malicious requests to SQL servers on any system, triggering a buffer overflow that was the basis for this worm's launch. Subsequent versions of IIS are now "locked down" [9], such that software has to be manually enabled to be accessible.

## 4.4 SUMMARY

This investigation of the UNIX and Windows protection systems shows that it is not enough just to design an operating system to enforce security policies. Security enforcement must be comprehensive (i.e., mediate completely), mandatory (i.e., tamperproof), and verifiable. Both UNIX and Windows originated in an environment in which security requirements were very limited. For UNIX, the only security requirement was *protection* from other users, and for Windows, users were assumed to be mutually-trusted on early home computers. The connection of these systems to untrusted users and malware on the Internet changed the security requirements for such systems, but the systems did not evolve.

Security enforcement requires that a system's security mechanisms can enforce system security goals even when any of the software outside the trusted computing base may be malicious. This assumption is required in today's world where any network request may be malicious or any user process may be compromised. A system that enforces security goals must implement a mandatory protection system, whereas these system's implement discretionary protection that can be modified and invalidated by untrusted processes. A system that enforces security goals must identify and mediate all security-sensitive operations, whereas these systems have incomplete and informal mediation of access. Finally, a system that enforces security goals must be tamperproof, and these systems have unbounded TCBs that provide many unchecked opportunities for untrusted processes to tamper with the kernel and other TCB software. When we consider secure commercial systems in Chapters 7 and 9, we will see that significant changes are necessary, but it is still difficult to undo fully the legacy of insecurity in these systems.

[9] Features that are not required are disabled by default. Bastille Linux performs a similar role to lock down all services in Linux systems [20].