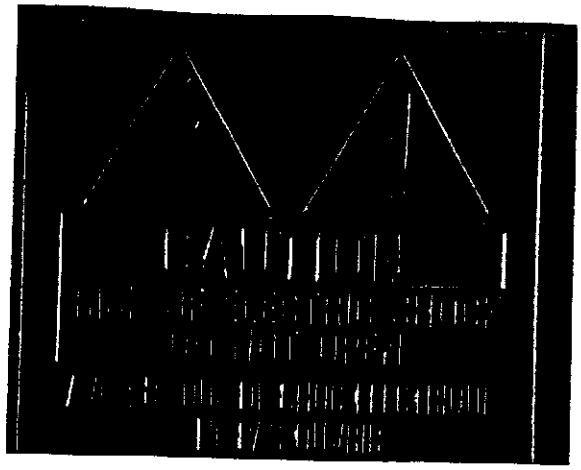


Unix and Windows NT security models have their advantages and disadvantages. Neither offers clearly superior security.

John Viega and Jeffrey Voas

The Pros and Cons of Unix and Windows Security Policies

Supporters frequently tout Windows NT as being the most secure commercially available operating system. Others tend to believe this opinion after hearing of Unix's many infamous security vulnerabilities. In reality, the two operating systems have far more in common from a security point of view than people expect. This, then, makes it difficult to honestly assert that NT is more secure than Unix.

By providing a brief introduction to the security architectures of Unix and Windows, we hope to convince readers that both operating systems have substantial merit from a security point of view and that neither operating system offers clearly superior security.

BASIC OPERATING SYSTEM SECURITY MODELS

Although the security models of Unix and Windows are largely different, they are based on the same fundamental concepts and concerns.

Two main divisions

To begin, both Unix and NT systems are divided into user-level code (often called the *user space*) and an operating system kernel. Application programs typically execute in the user space but may occasionally make a call to the kernel if they need special services. Such services are said to be "running in kernel space." Kernels are where operating system developers typically implement security policies, which manage access to devices, files, processes, and objects. Figure 1 shows how this basic division works.

Interprocess protection

Applications aren't generally concerned with how the security is implemented; applications running in user space simply experience security restrictions that are implemented in the kernel. One of the most important of these restrictions is *process space protection*. This restriction ensures that a single process can't directly access the memory allocated to other processes. In addition, no process can directly access the memory in use by the operating system. Thus, the operating system's security mechanisms mediate all interprocess communication.

Interestingly enough, interprocess protection didn't exist at all in earlier 16-bit Windows releases. So it was often possible to change data in other programs just by exploiting a bug in a single program since all programs shared a single address space.

As part of these user-level protections, processes are also not allowed to directly access devices attached to the computer, such as hard drives, video cards, and so on. Instead, special utilities inside the kernel (device drivers) act as wrappers/partitions around these devices. User-level programs must make calls through the kernel to these device drivers to access hardware subsystems. Most frequently, they make such calls indirectly through a system call interface. For example, in Unix, devices appear to the application simply as files on the file system; the application communicates with the device by performing file reads and writes.

Microsoft did not originally design Windows 95 and 98 to afford the level of process protection

1520-9202/00/$10.00 © 2000 IEEE

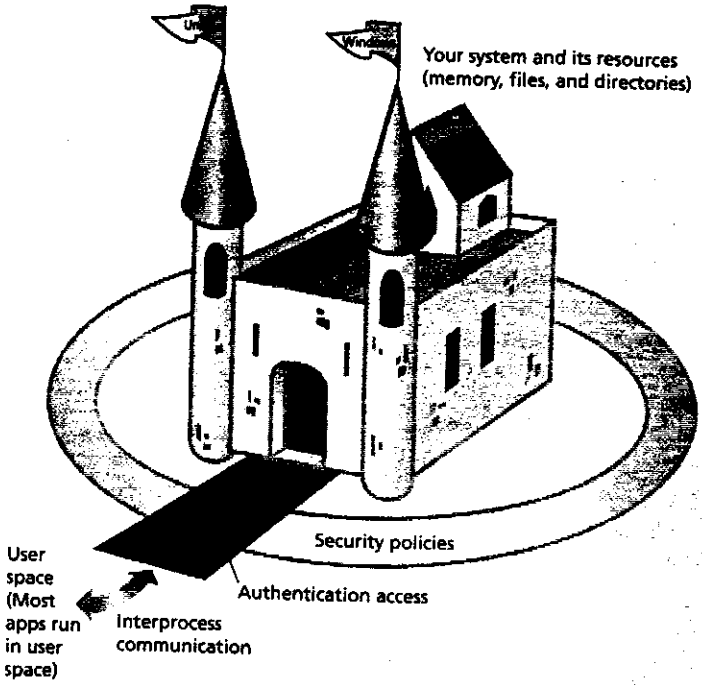
that more recent operating systems provide. These product lines are descendants of DOS (disk operating system), which Microsoft designed when security was not a significant issue—most personal computers were single-user devices that did not exist on a network. Although Microsoft has added some highly desirable functionality to more recent DOS descendants, they retain certain ingrained aspects of DOS that make it virtually impossible to build hacker-proof security policies. As a result, these additional features are more closely tied to improvements in reliability than to improvements in security.

Protection inside the kernel

Inside the kernel, there are usually no security checks. For example, no mechanisms protect one part of the kernel from other parts of the kernel; all kernel parts are explicitly trusted. This trust is usually extended to code that is not part of the operating system but executes inside the kernel (such as device drivers).

Thus kernels generally do not protect themselves from themselves. If a security defect exists in the operating system, anyone able to exploit that defect can exert complete control over the machine by using appropriate software applications. Building self-protecting kernels is difficult, and self-protection usually comes at a huge hit to performance, so kernels are infrequently built this way. Nonetheless, some operating systems—such as Unix-based Trusted Solaris—offer this sort of protection. To our knowledge, Microsoft does not offer any such self-protecting kernel for the Windows platform.

Figure 1. Unix and Windows share several fundamental concepts in implementing security policies.



Think of your system and its resources—things like memory, files, and directories—as the grounds and items within a castle's walls. The castle's walls and the moat represent the *security policies*, which both Unix and Windows implement in the operating system's kernel. Among these policies are those for *authentication* and *access*, which sort of act as the drawbridge to what's inside the castle; this drawbridge is the vehicle for *interprocess communication*, which is how applications running in user space access a system. Just like the castles of old, the kernel provides good protection from intruders outside itself, but not those arising from within.

AUTHENTICATION

A basic function of any security system is to provide *authentication*, a process which determines if a user's login information is valid—that is, if she is the correct person to have access to the system.

Operating systems provide techniques for securing user authentication. Such techniques differ from network-based authentication, which implements security via the program that provides the network service (though the program may leverage the operating system's built-in authentication mechanism).

Unix system authentication

Most Unix systems base authentication on user names and passwords. The system keeps a password database (usually in the file /etc/passwd). Since it is common to use the same password for multiple accounts on multiple machines, it is not desirable to let system administrators have access to user passwords. As a result, the database does not store actual passwords; instead, it stores a cryptographic hash of the password.

Hashing passwords. A hash is simply a one-way transformation of the password into another word. When it comes time to authenticate a password for a user attempting to access a machine, the user types in his password, which the system again hashes in the same manner that it used to store the password. If the two hashes match, the system authenticates the user and gives him access.

Unix systems have traditionally used a hashing algorithm based on the US government's Data Encryption Standard (DES). As a result, Unix systems implement the algorithm

so that it considers only the first eight characters of a password, discarding the rest. Consequently, attackers can conceptually launch a brute-force attack with a dictionary of all eight-letter passwords. Such an attack was once infeasible. Today, with ever-faster hardware and distributed computing, a brute-force attack is feasible for attackers with sufficient resources. Perhaps in a few years, this kind of attack will take only a few days using an average desktop PC.

This situation points to the need for password protection schemes that do not ignore the extra information. In fact, some Unix systems have recently begun switching to the MD5 hash function, which allows arbitrary length passwords.

Storing passwords. There are also security issues related to the database that stores passwords. When a system uses Unix's standard Network File System (NFS) software to maintain its password database, it is difficult but possible to circumvent authentication and thus gain access to an individual machine by controlling network access. The reason such break-ins are possible is that the NFS protocol does not provide adequate protection for password information as it traverses the network. A savvy attacker can forge responses from an NFS server and gain access even with an invalid password.

Attackers can use a flaw in one part of the operating system to attack another part of the operating system.

Additional Unix security prevents attackers from keeping a catalog of common passwords that would let them look up a password based on the resulting hash. This mechanism is called a *salt*. The system chooses random data and appends it to the front of the password—"salts" the password—before hashing it. So even if two people have the same password, the corresponding hashes should be different since each person is likely to have a different salt.

There is one weakness with this scheme: Unix systems store the salt alongside the password. If attackers have the password database, they can attempt to hash every password in a dictionary of common passwords by using a particular salt. They can then check for matches with the stored hash. This scenario is another brute-force attack and is remarkably effective at breaking real passwords in only hours or days.

Authentication on Windows systems

Windows uses the same concept of password hashing as Unix. However, the details behind the hashing algorithm and the hash storage approach are different.

Windows NT employs two different types of hashes. The one most commonly used is called the LAN Manager password. System administrators usually use this one because it interoperates with older systems (such as Windows 95). Unfortunately, however, this hash is relatively easy to break because passwords are not case sensitive—the system converts all characters to uppercase before transforming them.

Another technical problem with this algorithm is that passwords are easily broken with brute-force attacks. For example, running l0phtCrack—a cracker tool—for little more than a week will likely try every password in a database. Further, attackers can use this tool to break into 90 percent or more of those databases within a few hours.

When a Windows-NT-based system is attached to a network, it is susceptible to the same types of attack as those based on NFS authentication. However, the default network authentication mechanism in Windows 2000 uses Kerberos, an encryption technology that helps prevent these types of attacks. Unfortunately, when authenticating to a non-Kerberos-enabled machine, the protection does not apply. Similarly, if a machine without Kerberos tries to authenticate to a Windows 2000 machine, the latter will use traditional, unsecure techniques for authentication.

ACCESS CONTROL

After a security system authenticates a user, it must typically enforce certain restrictions or privileges associated with access to system resources.

Unix access control

Unix differentiates users by assigning them a unique integer—a *UID* (user ID). Users can also belong to *groups*, which are simply collections of users formed to collaborate on projects. Similarly, each group has its own *GID* (group ID) identifier.

UID 0 is a special integer identifying the user that rules (administers) the system. This special identifier provides a user who knows the corresponding password with complete access to the entire machine.

The system assigns all objects within it (including files and directories) a UID and GID; these identifiers define who owns what objects. Along with the integers that define object ownership, the system associates access permissions with each object; these permissions indicate who can read from, write to, and execute the object (if it is a program).

Permissions. Each file has three sets of these access permissions. The first set identifies the user who owns the file. The second set defines the group that owns the file. The third set defines other "non-owner" users that have access rights granted to them by the owners. The exception to these permissions is UID 0, which can perform any operation on any object.

Typically, when a user executes a program, the system assigns the executing program and all its children processes the UID of the user running the program. This assignment helps manage access control to system resources. Objects

are always accessed from running processes. When a process requests access, the operating system looks at its effective UID (EUID) to determine whether to grant the request.

Generally speaking, the EUID is the same as the UID for a specific process. There are exceptions, however—some programs need access to special resources and can therefore change the EUID of a process during runtime to the UID of the user that owns the executable object. Such programs are called *setuid programs*. The owning user must explicitly mark executable files as setuid for this change to occur. This operation does not change the real UID associated with the process, which can thus return to its original permissions when special access is no longer necessary.

If the setuid utility contained a security flaw, it would be possible to run the flawed setuid utility on any system program file and gain access to resources that the program's owner has access to. Because setuid programs are usually owned by UID 0, flaws in them can easily lead to a compromise of the entire system.

This is problematic because in practice, attackers commonly break into a machine—usually by compromising the password in some way—using a less privileged account than UID 0. From there, attackers can exploit broken setuid programs owned by UID 0 to complete their break-in.

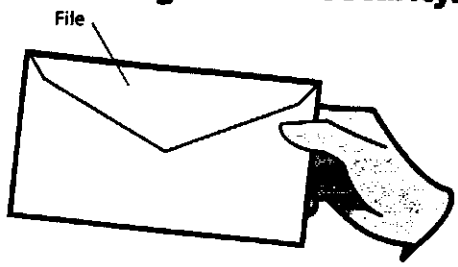
This demonstrates an interesting security problem: Attackers can use a flaw in one part of the operating system to attack another part of the operating system. Today, some Unix operating systems, such as Trusted Solaris, provide facilities to protect the operating system from itself. These operating systems are far better compartmentalized, which provides support for protecting one operating component from another.

Mandatory access control. Such facilities are usually packaged using a concept called *mandatory access control* (MAC), which is not found in all Unix operating systems, however. Figure 2 illustrates the difference between a typical operating system and one that employs MAC.

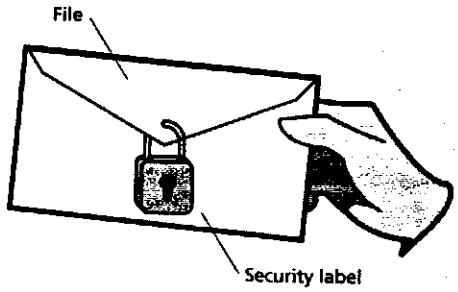
In a typical operating system, if a user owns an object (users can transfer file ownership of files they own, and UID 0 can arbitrarily set file ownership), that user can assign other users access to the object without restriction. Therefore, if a user owns a classified document, nothing prevents her from sharing the file with people who are not supposed to access classified information.

MAC solves this problem by introducing security labels to objects; these labels always propagate across ownership changes. Trusted operating systems use this function to prevent arbitrary operating system components from breaking the system security policy in handling objects created by other parts of the system. For example, Trusted Solaris's MAC makes it possible to prevent anything in the operating system from accessing the disk driver. In a typical operating system, attackers could circumvent any file protection mechanism imaginable by tricking any part of the operating system into running code that directly accesses the disk.

Figure 2. Mandatory access control (MAC) lets some Unix systems provide tighter file security.



In a typical operating system, if a user owns access to a file, she can grant access to whomever she chooses.



In an operating system with MAC, a security label—which specifies who has access rights—accompanies each object when it changes hands.

Although some Unix variants provide MAC, there are no versions of Windows NT that implement it. As a consequence, to get a system certified as B-level secure under the US Department of Defense's "Orange Book" criteria, it must not use Windows altogether.

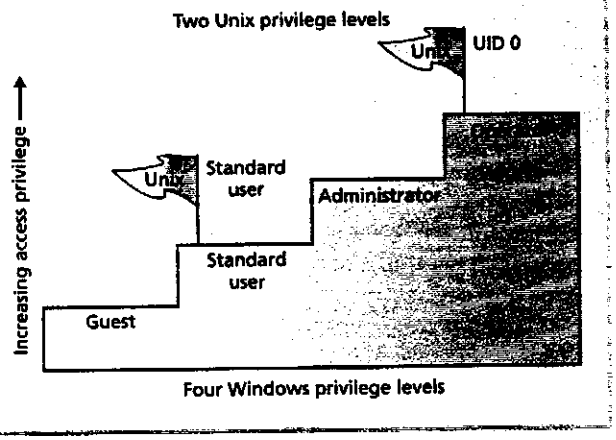
Windows NT access control

Much like Unix, Windows employs the notion of user IDs that are assigned to both individual users (account system IDs or SIDs) and groups (group SIDs). Windows has other concepts that do not directly map to the Unix model.

Tokens. The first such concept is the token. Windows NT has several types of tokens. The most important is the *access token*—a bit of data held by the machine that establishes whether or not the system has previously authenticated a particular entity. The access token contains all relevant information about the authenticated entity's capabilities. When deciding whether or not to allow a particular access to occur, the security infrastructure consults the access token.

Another important Windows token is the *impersonation token*, which allows an application to use another user's security profile. This token affords the same type of functionality as Unix's setuid programs—for example, imper-

Figure 3. Windows NT has a flexible access permission structure with several useful levels of privilege.



sonation has to be explicitly enabled for an application—but is implemented quite differently.

Basic security attributes. A second important concept in the Windows access control model is *security attributes*, which the system generally stores in access tokens. These attributes specify privileges that entities can be granted access to; system administrators use them for many purposes, such as recording whether the system can transfer particular rights to other users.

One of the most noteworthy differences between access control in NT and in Unix is that NT generally offers more granular privileges. For example, NT implements the right to transfer file ownership as a separate attribute; on a Unix system, file transfer rights are implicit in the notion of ownership.

File permissions. Similarly, file permissions are far more granular in Windows NT than for Unix systems. In NT, permissions are composed of a set of basic capabilities—such as the ability to read or the ability to transfer ownership. Unix provides only read, write, and execute permissions.

Windows NT offers four "standard" permissions although the system can create an arbitrary number. *No access* is one standard permission; it affords a user no access whatsoever to a resource. In fact, the system may not even query external attributes such as "size of file" for this level of permission.

The second standard permission is *read access*, which enables three capabilities:

- querying basic file attributes,
- reading data in the file, and
- executing the file.

Change access is the next step beyond read access; it adds the ability to modify and delete files as well as the ability to display ownership and permission information.

The final default permission is *full control*. The next step beyond change access, it provides the ability to change file permissions and to take ownership of a file.

One final security feature of the Windows NT file access mechanism that differs prominently from that of standard Unix is the access control list (ACL). Whereas Unix operating systems usually implement access control by specifying properties on an "owner, group, other" basis, NT again provides finer grained control.

Along with each entity protected by access control, the operating system stores an ACL—a list of users and groups with associated capabilities. For example, if you wanted to work on a collaborative project with Alice, Bob, and Chris, you could give Alice full access to all the files that belong to the project, while Bob would only have change access to those files. And you could also limit Chris's change access rights to a subset of the files. For other files, you could give Chris no access.

In contrast, traditional Unix systems do not offer such sharing. You would need to put the files under group ownership; place Alice, Bob, and Chris into a group; and give them all equal access to the files. So in short, Windows NT offers arbitrary flexibility without having to use the notion of a group.

NT has seven default directory permissions. Unix has only three: read, write, and execute.

Other permissions. Another significant advantage of the NT permission model is that the permissions structure is not as "flat" as that of Unix. In a Unix system, UID 0 must own most interesting services (such as network services) because the kernel owns the most powerful capabilities. Unix also doesn't have UIDs more privileged than the average user but less privileged than UID 0. In other words, Unix offers "all or nothing" privilege for access requests that are not related to file access. So you can either run code in the kernel (and thus access the object) or you cannot. Such a scheme enforces a good security practice: Always grant the minimum privilege necessary to complete a task.

Windows NT does not have the same privilege limitations inherent in most Unix systems. NT breaks the privilege structure into four types, as shown in Figure 3:

- standard,
- administrator (a sort of "super user" status),
- guest (similar to a standard permission but theoretically more restricted), and
- operator.

NT's administrator status is just as dangerous as Unix's UID 0, but the privilege type that sets NT far apart from Unix is the operator, which defines useful subsets of administrator privilege. An example is the print operator, which lets a service perform printer management tasks. The print operator privilege has only limited file access: It permits

only file writes and deletes to a single spool directory. Consequently, if attackers break into a program and obtain print operator access, they can damage the file system only on a single spool directory.

NT Server makes better use of the operator type than does NT Workstation. Machines running Workstation use only two operator types, which by necessity must incorporate—and hence, mix up—many types of functionality.

In Windows 2000, the granularity of rights assignment increases dramatically. Although the administrator user still exists by default, refinements to the Windows 2000 privilege assignment scheme makes an administrator account unnecessary.

A single virus will have a harder time infecting all Unix systems.

THE THREAT FROM REMOTE ATTACKS

Having multiple users requires some sort of remote access (after all, only one person can sit before a PC at once). The problems created by remote access have long been an Achilles' heel for Unix. Many of Unix's infamous security problems have occurred when it allowed improper access to remote, authenticated users.

In contrast, Windows was not designed as a multiuser operating system. While NT supports multiuser access, most machines running Windows support only a single user at a time, sitting at one machine.

For these reasons the Windows design reflects much less concern for inappropriate, remote access by, for instance, local users raising their privilege. Its designers were much more concerned with attackers breaking into a machine without an account. This emphasis occurs because local users generally have hardware access. A skilled attacker can take over a machine with hardware access alone and do so fairly easily; there's no need to exploit software flaws.

Another security concern with remote access is the threat posed by malicious mobile code. This problem received a lot of media attention, especially after the Melissa epidemic and, more recently, the I Love You virus. This type of virus is a significant problem for Windows architectures but it has not affected Unix machines whatsoever. The problem these viruses exploit is that Windows e-mail applications are easier to trick into running code that is part of an e-mail message.

Further, Windows-based mailers make it simpler for users to run code inside an e-mail message. It usually boils down to a single mouse click whereas in Unix, a user would have to perform many tasks to do the same.

Windows's lack of diversity in its product architecture also plays a role here. Today, Windows has essentially one e-mail application: Outlook. Therefore, a Windows virus can quickly execute a script that reads Outlook's address book on NT machines. But in Unix, there are dozens of popular mailers, and address books are less common. Therefore, a single virus will have a harder time infecting all Unix systems since Unix operating environments often vary drastically.

OTHER CONSIDERATIONS

The single most widespread cause of security problems in the past three years is buffer overflow. Buffer overflows are as easy to write for Windows NT as they are for Unix. The core problem is that C does not check buffer boundaries, so you cannot blame operating systems for this situation.

A final difference that we should mention is that Windows's source code is not publicly available (as is that for Unix). Thus, fewer people are looking for problems in its source code. As we all know, source code inspections are a good way to find problems, but nothing prevents "bad guys" from also using source access to find security problems. On the other hand, source code availability or lack thereof never stops clever hackers: They simply reverse engineer the code and break it just as if the source had been available.

You should now realize that both Unix and Windows NT have their advantages and disadvantages. NT's biggest shortcomings tend to be poor user authentication, a susceptibility to malicious mobile code, and the fact that mandatory access control is impossible. For Unix, the biggest shortcoming tends to be its inflexible privilege system.

In summary, do not rely on an operating system as your sole line of defense. While both Unix and Windows employ reasonable security models, neither is sufficient. It is up to you to keep abreast of security-related press releases and newly released patches. By being aware of your security risks and state-of-the-art tools for mitigating those risks, you will be as well equipped as possible to fight these ever-present dangers. ■

John Viega is a software security consultant at Cigital (formerly Reliable Software Technologies) in Dulles, Va. Contact him at jviega@cigital.com.

Jeffrey Voas is chief scientist at Cigital. Contact him at voas@cigital.com.