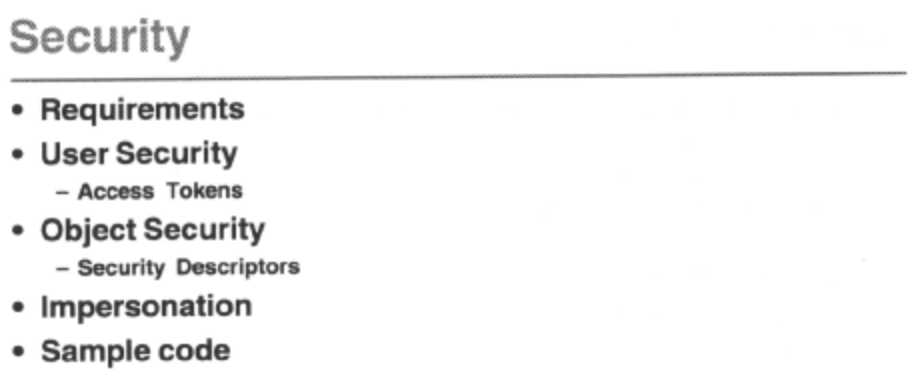


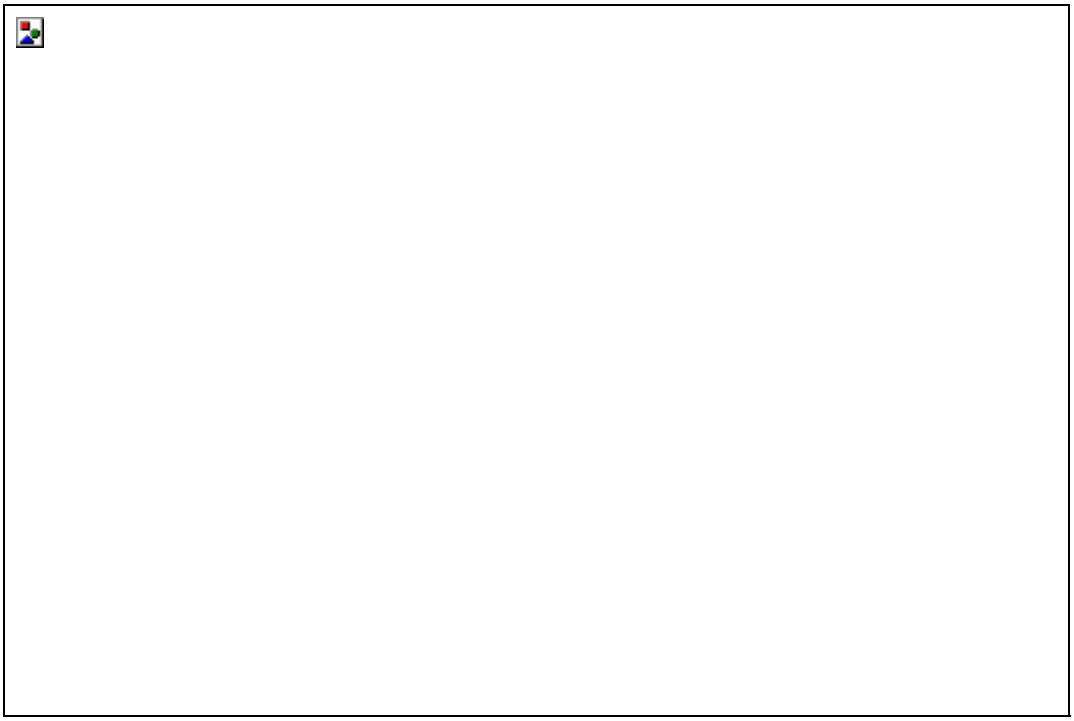
Introduction

A secure multi-user multitasking operating system must protect processes from adversely affecting each other or the system; part of this requirement involves protecting one user's files, memory and other resources from unauthorized access by another user. It also means protecting the operating system's files arid memory from user programs. It also requires that any attempts to bypass the security features are monitored and maybe even alarms raised.
The *security features* of Windows NT pervade the Win32 API and can optionally be used to provide a number of additional services in a Win32 application. A security-aware application could, for example, allow the user to query the security attributes of a file, provide detailed feedback when access to a secure file is denied, or customize the security attributes of a file or group of files so that only a subset of other users on a network can access the information. Also, any application that manipulates system-wide resources (for example, system time) must use the security system to gain access to that resource.

Even if you are not planning to make use of these features in applications that you write, an understanding of Windows NT security makes the behavior and administration of the system easier, lessening the chance of making mistakes, which could compromise sensitive data.

Objectives

- By the time you have completed this chapter, you should be able to:
- List the requirements of C2 Level security
- Explain the details of user-based security, involving user logon and access tokens
- Understand object-based security using security descriptors
- Describe what happens when a process attempts to open a handle to an object
- Explain the basic concepts of privilege and impersonation
- Write a simple program to attach a security descriptor to an object



NT supports C2 level security, as defined by the US Department of Defense. C2 level security includes the following requirements:

The owner of a resource must be able to control the access to that resource and what can be done with it. This access control must include or exclude individual users or named groups of users.

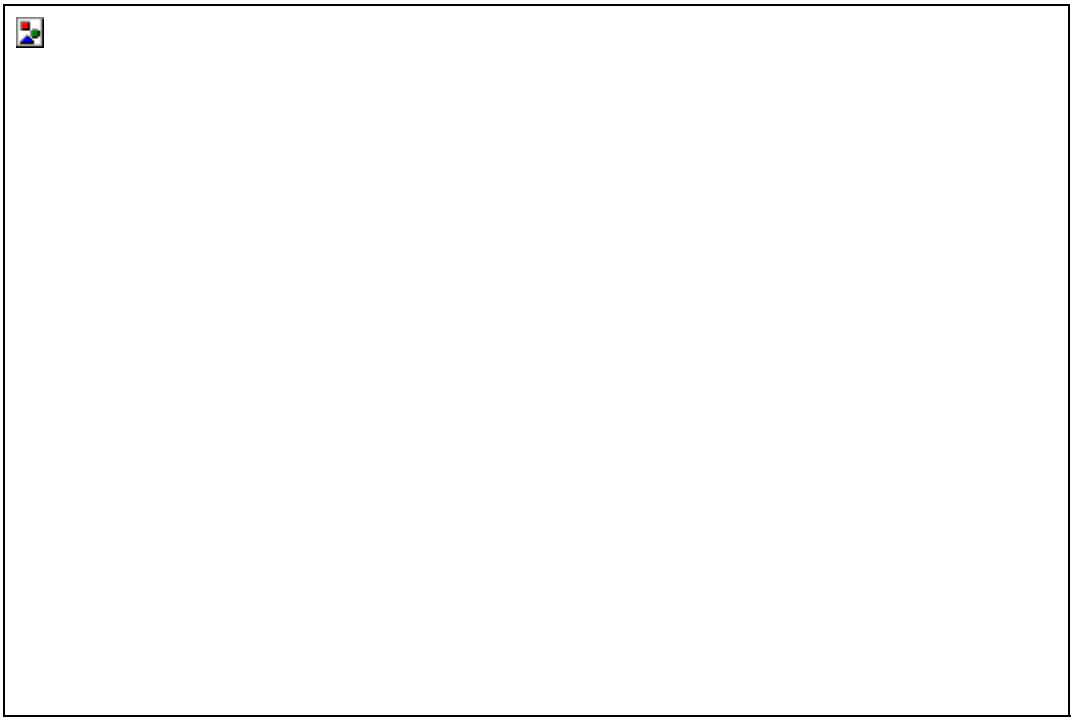
Memory must be protected, so that its contents cannot be read after it is freed by a process, i.e. it must be re-initialized.

Users must identify themselves with a unique identifier and a password when they log on. All auditable actions taken by the user must be associated with an identifier for the user.

System administrators must be able to audit security-related events. Access to this audit data must be limited to authorized administrators.

The system must protect itself from external interference or tampering.

NT will support B 1 level security in future releases. B 1 security includes all the requirements of C2 security and adds mandatory access control and data labeling. These additions will make it possible to prevent a user who has access to protected information from supplying that information to a user who does not have the required clearance.



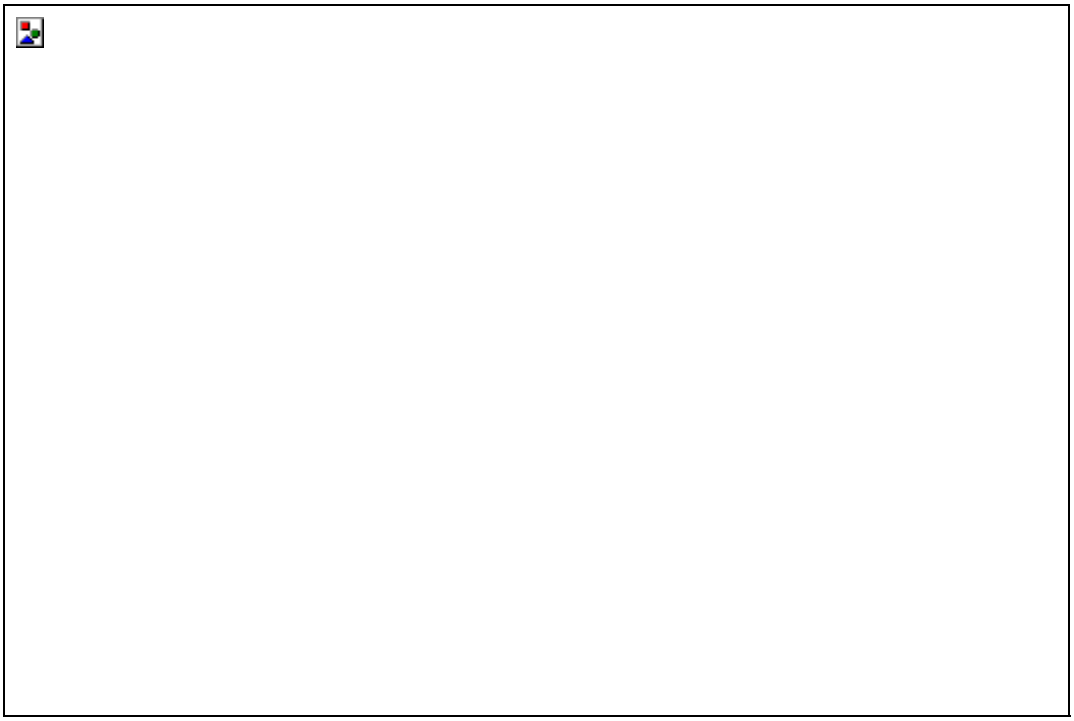
Windows NT security authentication and validation is dealt with by 3 components; the *Security Subsystem,* the *Security Reference Monitor* and the *Logon Process.* Whether the security system will grant a user access to an object depends on the security information associated with both user and object.

A *user* is a person that logs on to the system using a password and who is subsequently authenticated. Users can be organized into *groups, which* have the same profile. It is not possible to log onto the system as a group, they simply exist as a convenience for resource administration.

An *object* is a resource that can be accessed by a user.

The first line of protection in Windows NT is *authentication;* to validate each user's identity. Every user is therefore required to logon to the system before using it. If the logon is successful, an object called an *access token* is permanently attached to any process representing the user. This access token identifies, amongst other things, who the user is and what groups he/she is a member of. Users and groups are identified by unique *security identifiers,* issued when the user/group account was added to the system. The access token is used to identify the user whenever a process subsequently attempts to access an object on behalf of the user.

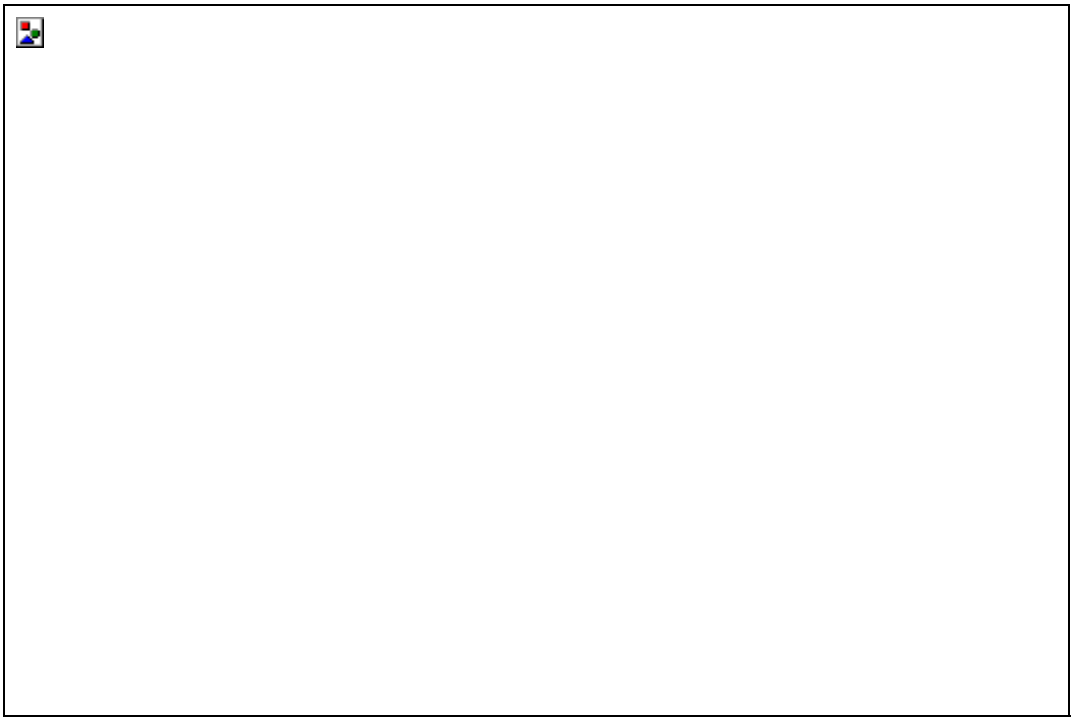
Object protection is the essence of access control and auditing in Windows NT because all system resources that can be compromised are implemented as securable objects. A securable object is an object to which a *security descriptor* can be attached, detailing which users/groups can/cannot access this object, and what they can/cannot do with it. All named objects are securable and some unnamed objects are too, for example, processes and threads.



When a user process creates a securable object, it can specify who else can use the object and what they can do with the object. Any Win32 API function that creates a securable object provides the option to assign *security attributes* to it.
The Window NT security system takes the security attributes and attaches a security descriptor to the new object. If no security attributes are specified when an object is created, the security system attaches a default security descriptor to the object.

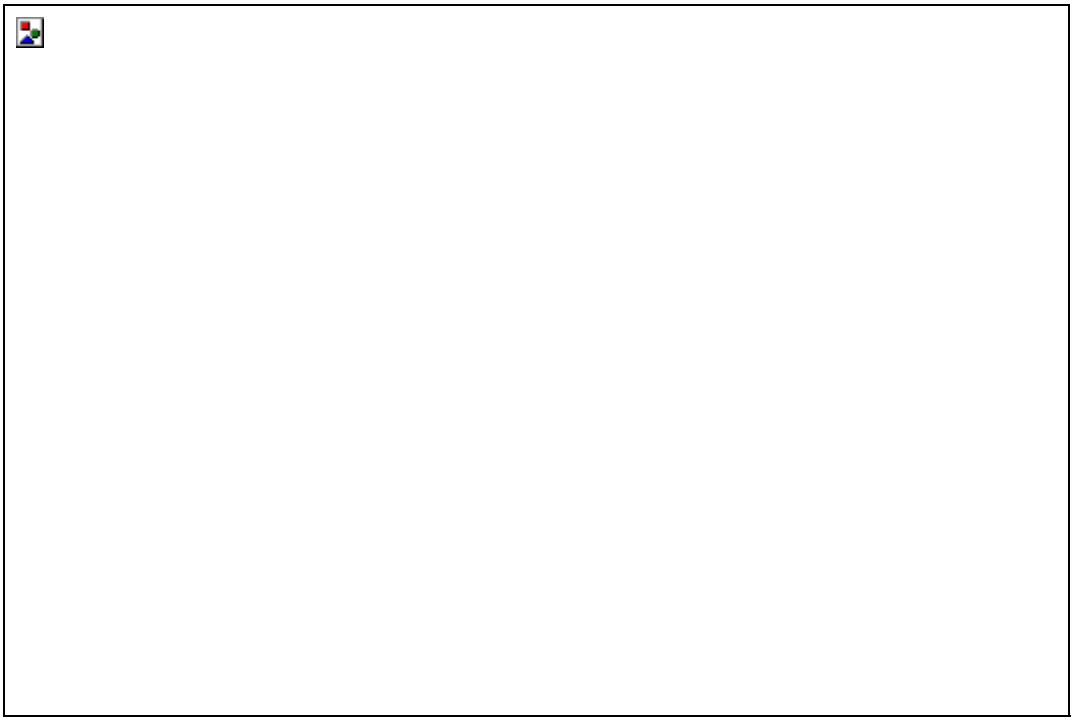
Another process must have a handle to an object before it can access it. When attempting to open a handle to an object, the process must specify what it wants to do with the object, in other words, its *desired access rights.* The Object Manager in the NT Executive calls the Security Reference Monitor.

This compares information in the user's access token against the object's security descriptor to determine whether the process is allowed to use the object. If so, it returns a handle that contains the *granted access rights* to the object. On subsequent attempts to use this handle, the Object Manager simply compares the granted access rights for the handle with the type of access implied or specified in the API call.



Security information is held at two levels in Windows NT. The object owner applies object security, usually when the object is created. This defines who can access the object and what operations they can carry out. The object owner can modify this security information, allowing or denying access to users on the system.

The administrator of a machine may grant privileges to users. These privileges override object security, allowing a user to perform operations that would normally be denied by object security. The most powerful privilege is the ability to change the owner of an object. Once a user has gained ownership of an object then that user can then modify the object security permissions.



Each user, or group of users, is assigned a *security identifier (SID)* when the user/group account is added to a particular security domain. A domain is a set of networked servers, logically grouped for administrative purposes. SIDs are unique across all security domains and once a SID has been used to identify a user or group account, it cannot be used again at any time to identify any other user or group account. There is one exception; the only SID that is not guaranteed to be the same from logon to logon is the logon-identifier SID.

A SID is not simply an ASCII name but a value of variable length that uniquely identifies a user or group. It consists of a *revision level*, an *authenticating authority* value (the SID issuer, typically Window NT), a set of *sub-authority* values (typically representing the network domain that will be the user/group primary domain) and a *relative ID* (RID), which is unique within the authenticating authority/sub-authorities combination (typically representing a particular user or group on a domain). The combination of the sub-authority values and the RID can be thought of as a RID which is relative to and unique within, the authenticating authority. A typical SJD looks like this:

5-1 -000005-15-251 7fc4- 1 3c43ba3-4dc74cb-ffffffff

Each SID-issuing authority issues a given RID only once, so joining these values ensures that no two SIDs will be the same, even if two different SID-issuing authorities issue the same RID. Because a primary domain identifier is coded into a SID for a user/group account added to a domain, switching domains causes the creation of a new SID to represent the user/group. This will cause the user/group to lose much of the access previously afforded them.

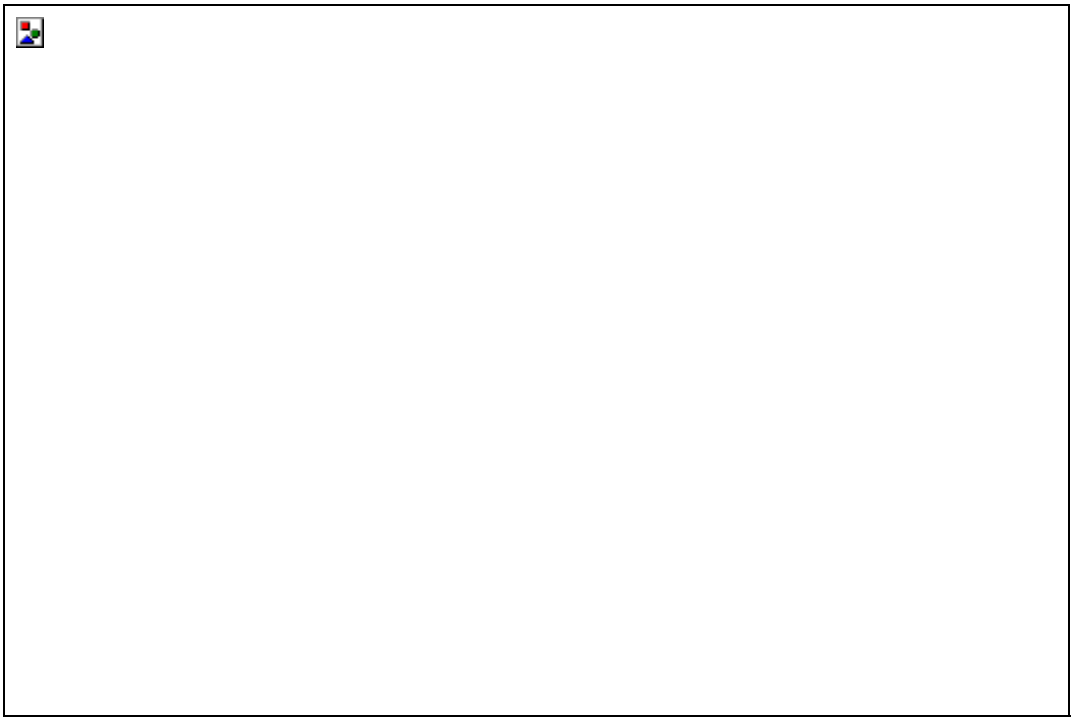
Deleting a user account and then adding a new user account with exactly the same account details will result in a new unique SID. However, changing the profile details within an account does not affect the SID.

By using SIDs for user and group identification, there is unambiguous user and group identification, portable from one security domain to another.

Some accounts and SIDS are pre-defined and are automatically a part of every installed system. A *universal well-known SID* is meaningful on all secure systems using this security model, including systems not running Windows NT. For example, S-1-1-0 is the 'World' SID (SECURITY_WORLD_SID_AUTHORITY and SECURITY_WORLD RID) and always identifies the special group account that includes all users. An *NT well known SID* is not universal but is meaningful on all Windows NT installations.

For example, S-l-5-12 is the 'local system' SID (SECURITY_NT_AUTHORITY and SECURITY_LOCAL_SYSTEMRID) and always identifies the special account used by user-mode parts of the operating system. Some accounts are relative to each domain. For example, S-I-5-....-1f5 is the 'guest' SID (SECURITY_NT_AUTHORITY and DOMAIN_USER_RID_GUEST) and always identifies the special group account that can be logged onto by users that don't have an account.

Note that security identifiers are not directly addressable by applications, but can be queried or manipulated only through Win32 API functions. For example, if an application has an account name and needs to know its SID, or vice versa, it can use the API functions `LookupAccountName()` and `LookupAccountSid()`.



To understand who you are when you log on, the system maintains a database of all the users that have accounts in each network domain. This *Security Account Manager* (SAM) database contains, amongst other things, user names and passwords.

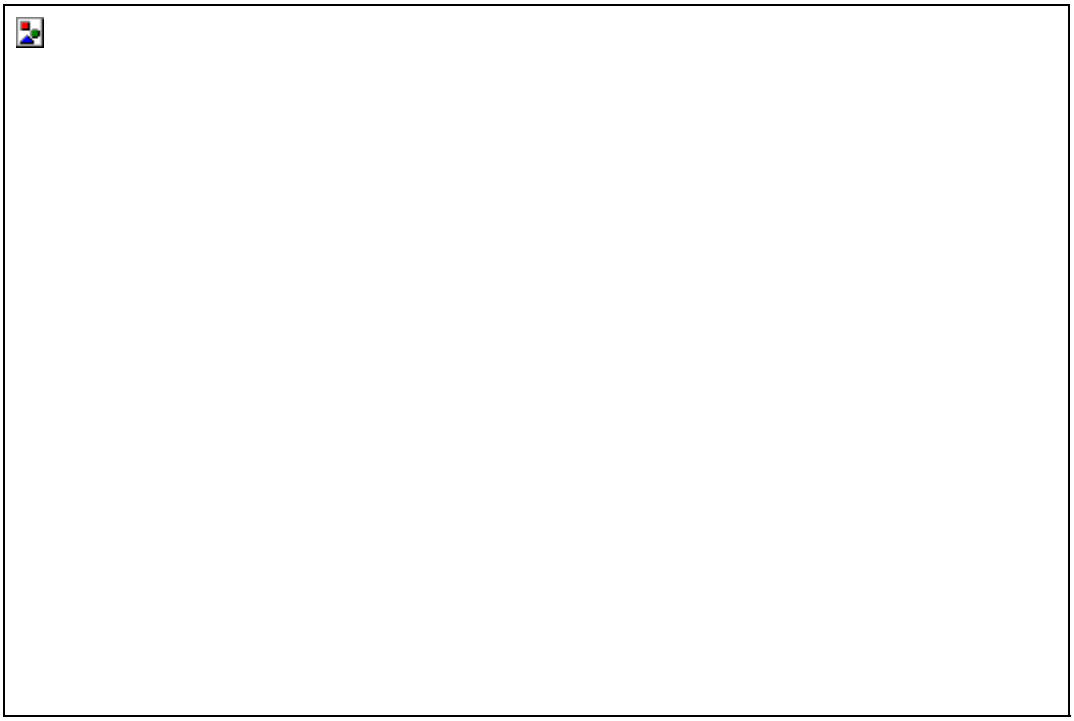
When you logon, either locally (interactive) or remotely (over a network), a logon process prompts the user for his or her account name and a password, which it passes to the *Security Subsystem.* The Security Subsystem uses the appropriate *Authentication Package* to verify the user's identity in the SAM.
If the account is not local, then the logon request is forwarded to a remote authentication package. Windows NT supports multiple authentication packages implemented as DLLs, which allows developers to introduce custom authentication routines that meet specific requirements.

If logon is successful, you become a *subject,* and an *access token* is returned to the logon process to represent the subject. The access token contains the subject's SID and other security information, such as groups the subject is a member of and privileges the subject has.
The SID is used to identify a subject uniquely across all domains on the network. Finally, the logon process calls the Win32 Subsystem to create a process to represent the subject and attach the access token to it. For an interactive logon this would be the shell process, the Program Manager.
Each Windows NT process is associated with a subject. Even Windows NT itself is a special subject that is always present, allowing the parts of Windows NT that run in user mode to have a security context when accessing objects (well-known SIDS). Windows NT is subject to the constraints of its own security!

This model supports both local and remote logons, thus numerous subjects may be logged-on at the same time, making Windows NT a multi-user system.

For local logons, the *Logon Process* (a Win32 application) waits for a user to press the Ctrl-Alt-Del key combination to log on. This key combination is securely trapped at a very low level and cannot be 'hooked'.

Remote logons and external requests to connect to a network resource or perform network **I/O** are supported by a local process interceding and verifying the access, using the services of the Security subsystem. The built-in Windows NT Server service or other network server services perform this task for network requests. For remote users, an access token is used to *impersonate* the user on the local machine, i.e. when handling requests from the remote user; it takes on the security attributes of that user.



An *access token,* created at user logon, is used to group security-related information on behalf of a user, some of which is shown in the diagram above. It identifies a user and the groups he/she belongs to, by SID Every Windows NT process has an access token attached to it which identifies the user to the operating system, since every request of the system is made in the context of a running process. Each process inherits a copy of the access token from its creating process. This association between a process and an access token allows fast access validation and allows each process to modify its security information in limited ways, without affecting other processes running on behalf of a user.

A user who has performed an interactive logon has the Program Manager as his/her shell process with an associated access token representing that user. The Program Manager is used to start other 'top-level' applications, which may then in turn create other processes. Therefore all these processes will represent the same user.

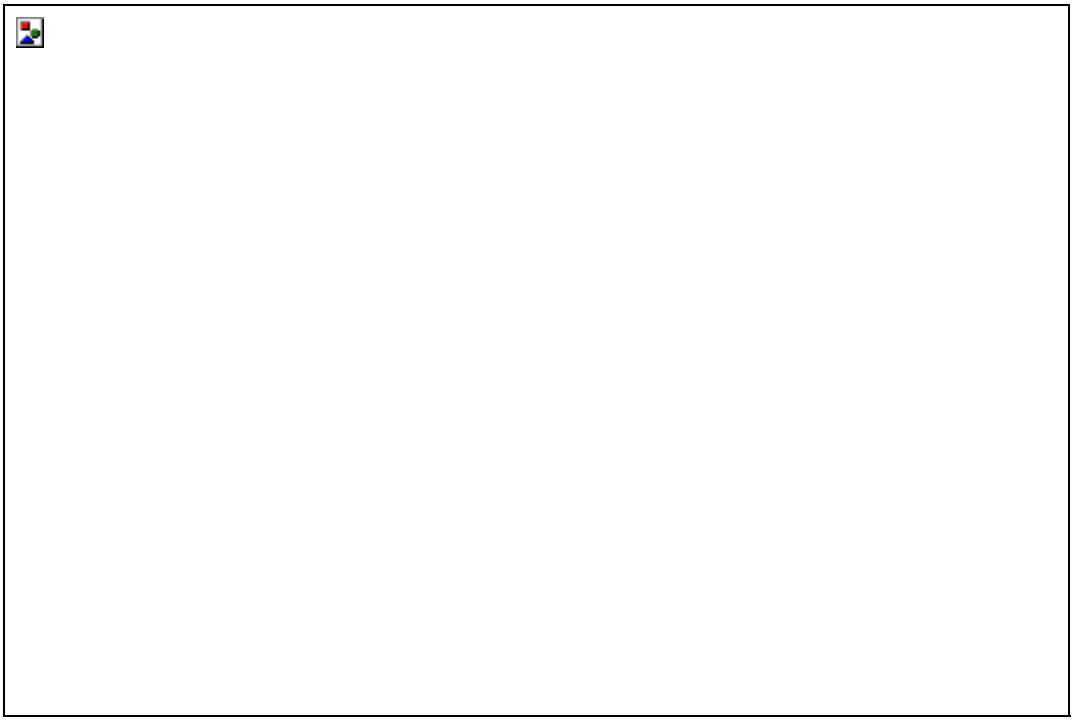
As we have seen, each user, or group of users, is assigned a Security Identifier *(SID)* when the user or group account is added to a network domain. SIDS are unique throughout the existence of an NT installation on a particular domain, so a SID uniquely defines a user or an arbitrarily large collection of users, in a format that is easy for the operating system to understand. The access token provided at logon is likely to contain SIDS for well-known groups like 'Everyone' (all known users) and 'Users' (all known users on the domain).

A *privilege* is a locally unique identifier used to regulate the use of some system services and system resources; they effectively allow security overrides. More on privileges later.

The *default owner* field of the access token specifies the SID that may be used as the owner of any objects created without security on behalf of the process represented by this access token. The SID must be one of the user or group SIDS already in the token. The owner of an object specifies who else can use the object and what they can do with it, and can perform almost any action on the object.

The *default access control list (ACL)* field of the access token allows the system to assign a default access control to the object if the creating process doesn't explicitly provide it. More on ACL later.

Note that access tokens are not directly addressable by applications, but can be queried or manipulated only through Win32 API functions. For example, to retrieve information in an access token, an application should call `OpenProcessToken()` to get a handle, and then `GetTokenInformation()`.



Knowing who you are is only part of the battle. The other part of the information the system needs to know is what you can do with a particular object, and this information is attached to the object. Each securable object is associated with a list that specifies users, and/or groups of users, by SID. and the access to the object that each user, or group of users is allowed or denied. By comparing this list with the access token of the process attempting to access the object, the system can ascertain whether you get the access to the object you want or not.

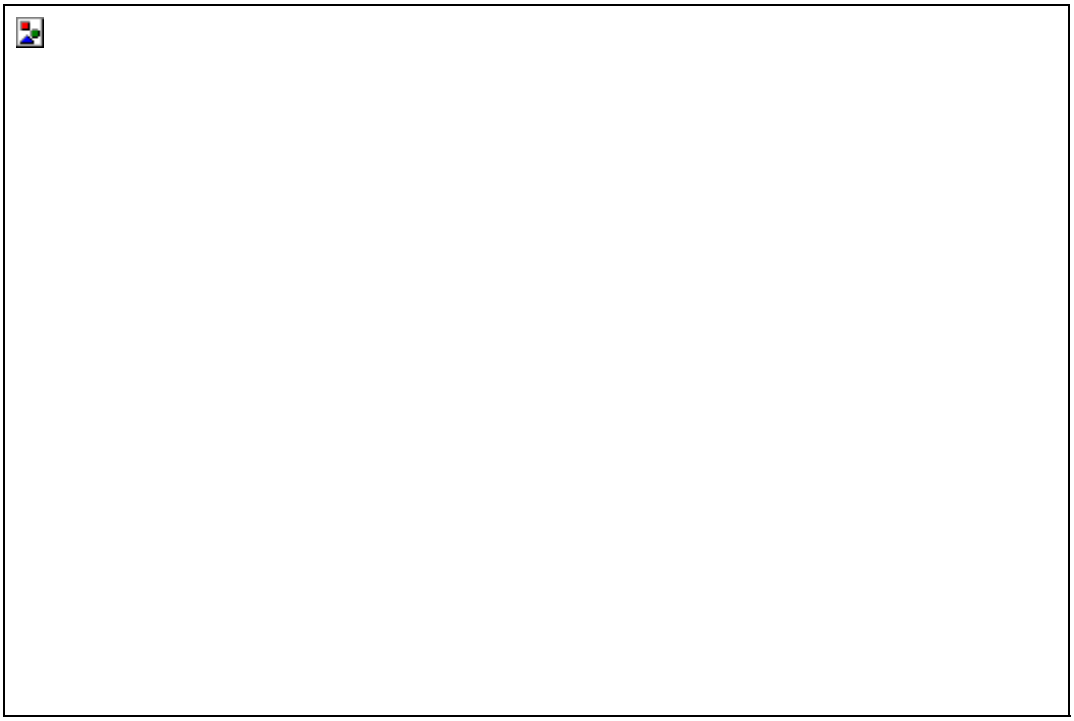
A *security descriptor(SD)* contains the security information that is associated with an object; precisely who (by SID) can do what with it. The *owner* in the SD is the SID of the owner of the object, who can grant discretionary access to the object and perform almost any action on it. The *group* in the SD is the SID of the primary group of the object and is mainly there for POSIX support.

The *discretionary access-control list (DA CL)* in the SD is the list, which grants and denies particular sets of accesses to the object to individual users and/or groups. The owner of an object controls the DACL. A DACL consists of a header and an ordered list of *access-control entries (ACEs).* Each ACE identifies a user (or group of users) and their access rights. When a process attempts to use an object, the system compares the security attributes listed in the access token with the ACEs in the objects DACL. The system compares the access token with each ACE until access is either granted or denied or until there are no more ACEs to check. Conceivably, several ACEs could apply to a token. And, if this occurs, the access rights granted by each ACE accumulate. For example, if one ACE grants read access to a group in an access token and another ACE grants write access to the user, who is also a member of the group, the user will have both read and write access to the object when the access check is complete.

The *system access-control list (SA CL)* in the SD is another list of ACEs, which specify which kinds of operations on the object, and by which user, should generate audit messages and alarms (future release). The SACL is controlled by a system administrator, and could allow him/her to audit unauthorized, or indeed any, attempts to gain access to an object. A process must have SE_SECURITY_NAME privilege in order to read/write a SACL for an object. This is to prevent unauthorized processes from reading a SACL and thus knowing what to do to avoid generating an audit trail or setting ACEs in a SACL to generate spurious auditing and thus cover their tracks.
Security Descriptors have two formats; *absolute* and *self-relative.* Absolute SDs maintains generic memory pointers to their constituent fields, self-relative SDs contain offsets to data appended to the end of the SD. The

reason for self-relative SDs is that SDs are required to be stored on disk and transmitted across network connections.

Like access tokens and security identifiers, security descriptors should be considered as opaque structures. There is comprehensive API support to manipulate security descriptors, e.g., `GetKernelObjectSecurity()`, `GetSecurityDescriptorDacl(), SetSecurityDescriptorSacl()` etc



Each of the entries in an ACL is an ACE which specifics what kind of access to an object by a specific user/group is allowed/denied, or specifies the types of access by a specific user that will generate system audit messages or alarms. A SID identifies the user/group. An access mask specifies the access rights being allowed, denied, or monitored.

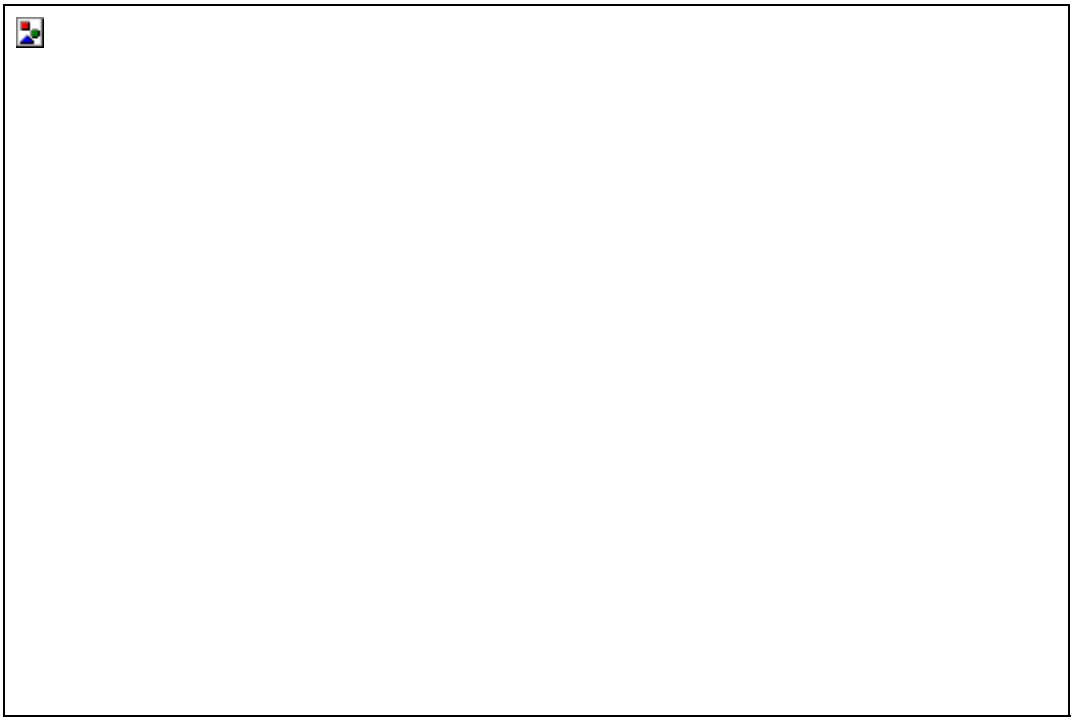
Discretionary ACLs have two *types* of ACE: ACCESS_ALLOWED_ACE_TYPE and ACCESS_DENIED_ACE_TYPE.
System ACLs also have two types of ACE: SYSTEM_AUDIT_ACE_TYPE and SYSTEM_ALARM_ACE_TYPE (*not supported in this release). A *flags* field describes how and if the ACE is to be inherited, and whether audit messages or alarms are to be generated for either failed or successful access attempts, or both.
Security auditing is used to keep a log of security-significant events, e.g. which users have accessed a secured file. A system audit type of ACE is used to generate security audit messages and cause them to be entered into a system audit log for later processing. An administrator can view this log by using the Microsoft Windows *Event Viewer*. The event log can also be manipulated by using the event logging API functions.

An *access mask* is a single 32-bit value that defines a particular set of abilities that can be granted or denied to a process when it attempts to use an object. For example, if a process attempted to write data to a secured file object but did not have FILE_WRITE_DATA access to the file, the system would refuse the attempt. It is the same as the desired access mask you pass to the Win32 API when you open or create an object. It contains the *specific rights, standard rights,* and *generic rights.*
The first 16 bits are the specific rights, which apply only to the object type associated with the access mask. E.g. bit 0 EVENT QUERY_STATE for an event object but FILE_READ_DATA for a file object.

Bits 16-23 are the standard rights, which apply to all objects. WRITE_DAC allows the process to modify the object's DACL and thus change the protection on it. WRITE_OWNER allows the process to transfer the object ownership to any SID in the processes access token. READ_CONTROL allows the process to query the DACL, owner and other SD information. DELETE access allows the process to delete the object. Note that although an object is deleted and no longer useable, the storage associated with the object is not freed until all open handles to the object are closed.

SYNCHRONIZE access allows the process to synchronize execution with some event associated with the given object that changes its signaled state. ACCESS_SYSTEM_SECURITY allows modifying of the SACL. This access cannot be set through the DACL but requires that the caller have SE_SYSTEMNAME privilege. MAXIMUM_ALLOWED cannot be allowed or denied, but can be set in a desired access mask to modify the algorithm used to scan the DACL, so that the object is opened using all the access rights that are valid for a given user.

The fact that object-specific rights are different for each type of object, makes them difficult to use. Logical '*read*', '*write*', '*execute*' and '*all*' access means different things to different objects, and thus different access bits. Bits 28-31 are the generic rights, which are mapped to a different set of specific and standard rights for each object; Windows NT
maintains a *generic mapping* for each object type. Generic rights arc useful when an application request access to an object, because they allow the application to avoid querying and setting the rights for a specific object type.



When an application creates an object, any access rights explicitly provided in a SD are assigned to the object. If the creator does not explicitly provide a SD, the system searches for default security information in different places, depending on the information required.

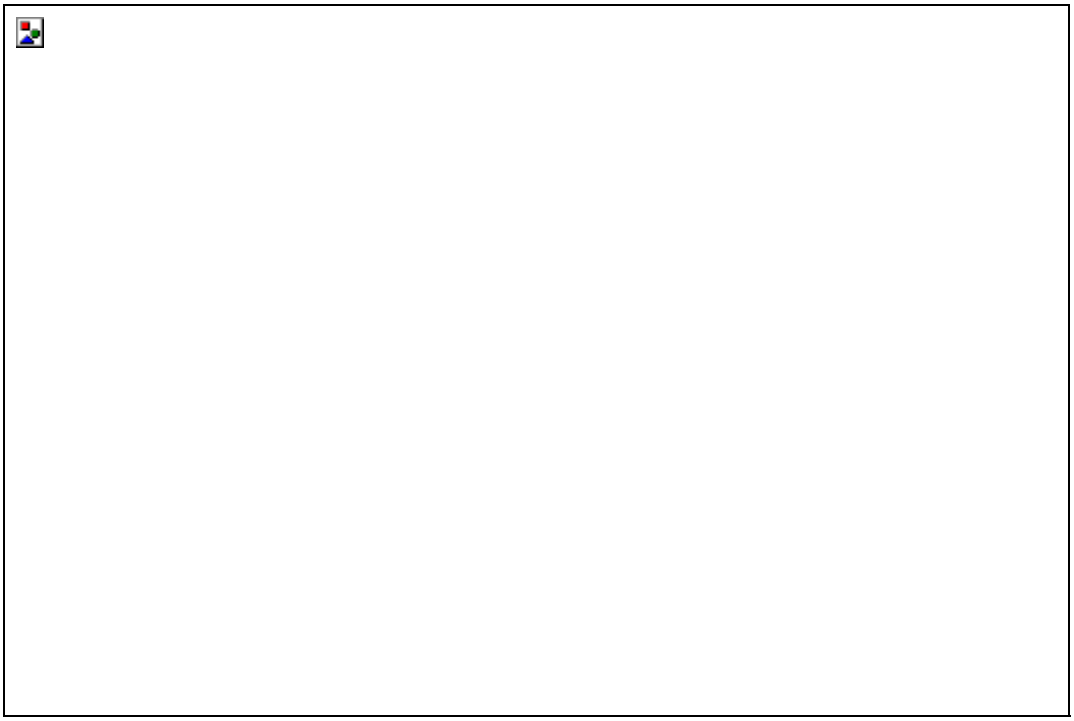
To assign an owner to a new object when a SD is not provided, the access token of the creating process is checked for a default owner SID that is assigned as the owner. If not found the SID from the access token of the creating process is used. Even when an owner SID is provided in a SD, the system checks these values in the access token to make sure the specified owner can be assigned ownership of the object.

The Object Manager supports a hierarchical naming structure much like a file system. This allows objects to be grouped in the object namespace and for *object domains* to be defined allowing the object namespace to be extended. For instance, the I/O Manager is a secondary object manager, looking after an object domain containing directory, file and device objects, all under a node of the Object Manager's namespace. To facilitate this there are two categories of object. A *container object* (or *object directory)* is one, which exists to logically contain other objects, e.g. a file system directory. A *non-container object,* using the above example, would be a file. This distinction is used to establish object protection inheritance rules.

When the creator does not provide a security descriptor, a DACL is assigned to a new object as follows.

For a named object, the discretionary ACL of the container object in which the new object is to be stored, is checked for inheritable ACEs and a DACL created from any found. Each ACE can be marked for any one of no inheritance, for inheritance by sub-container objects, for inheritance by non-container objects or for inheritance by both. So for example, protection on a file would be inherited from it's directory. This could however be different to the protection for a sub-directory inherited from the same directory.

If there are no inheritable ACEs, or if the object is unnamed, the system looks in the creators access token for a default DACL. If neither of these sources provides a DACL, the object is created without one, and universal unconditional access to the object is granted.



The Win32 subsystem uses Windows NT Executive objects to provide its own versions of these objects. For example, mutexes and events are directly based on NT Executive objects. In addition, the Win32 subsystem provides many more Win32-specific objects.

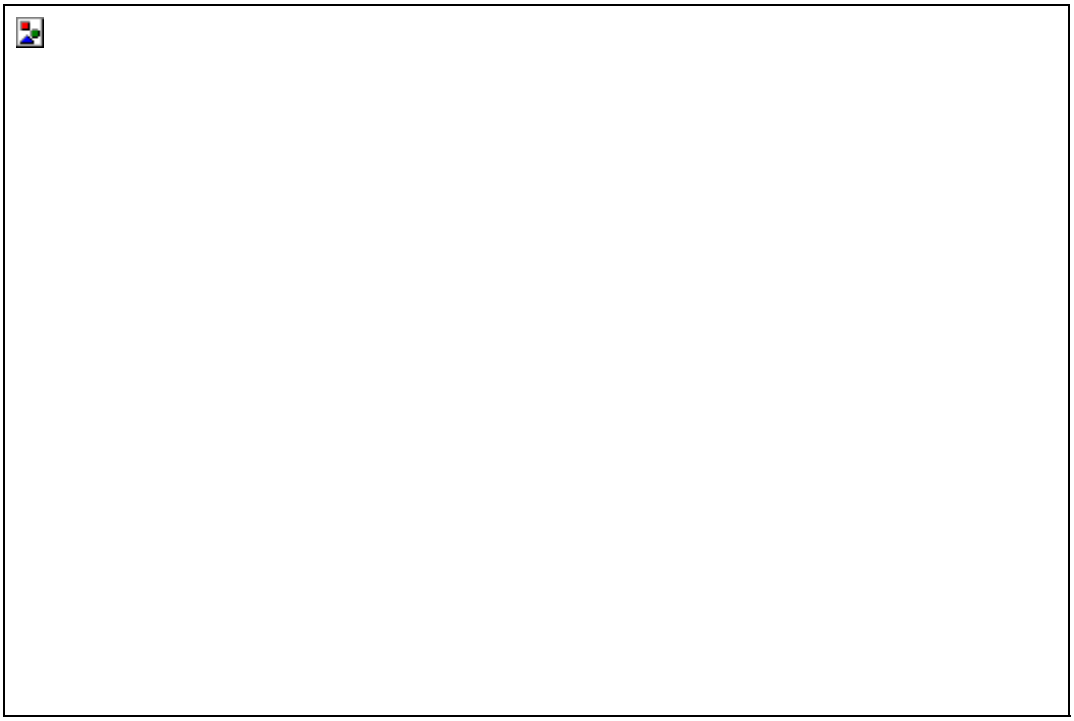
The kernel and window-management components of the Win32 subsystem support securable objects, i.e. security descriptors can be attached to them. On the other hand, there are no securable objects in the GDI component. Win32 API functions can be used to query and set the security attributes of securable objects. The following table lists some of the Win32 API functions related to Win32 securable objects:

Object type	**Related API functions**
`Kernel`	`SetKerne1ObjectSecurity()`
	`GetKerne1ObjectSecurity()`
`File`	`GetFi1eSecurity()`
	`SetFileSecurity()`
`User`	`GetUserObjectSecurity()`
	`SetUserObjectSecurity()`
`Private`	`CreatePrivateObjectSecurity()`
	`GetPrivateObjectSecurity()`
	`SetPrivateObjectSecurity()`
	`DestroyPrivateObjectSecurity()`
`Registry`	`RegGetKeySecurity()`
	`RegSetKeySecurity()`

The security features of Windows NT are available to Win32 applications automatically. Every application running on Windows NT is subject to the security imposed by the particular configuration of the local system. The impact of security on most Win32 API functions is minimal; a Win32 application that does not require any special security features does not need to include any special code.

It is possible for private objects to be registered with the system, created by a protected server process and accessed by multiple client processes. These private objects can be protected; the designer of these objects can decide what type of protection is required for an object.

NB Of the three file systems that Windows NT supports, FAT, HPFS and NTFS, only NTFS allows the placing of security information on files and directories as part of the file system. While messing about with security information on FAT and HPFS files will appear to work, it does nothing.



A process must have a handle to an object before it can access it. When attempting to open a handle to an object, the user's process specifies its *desired access rights.*

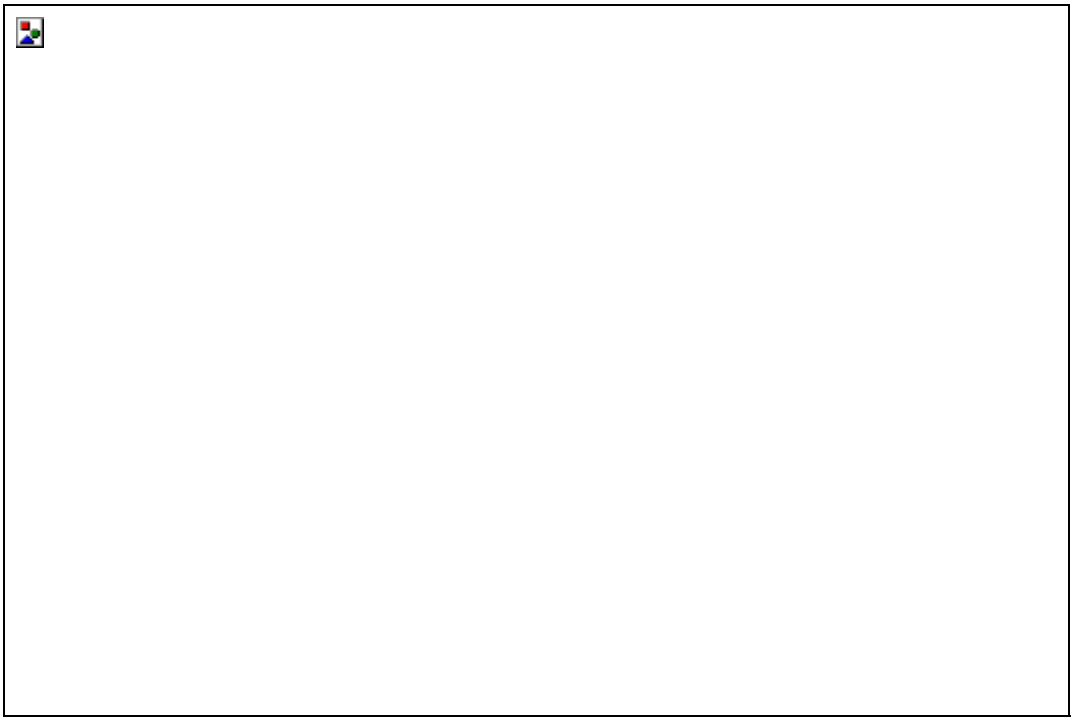
If the calling process requires to fiddle with the SACL and ACCESS_SYSTEM_SECURITY access is requested then the required privilege must be checked, and the request is rejected if not.

The security system walks the DACL in the SD for the object, from first ACE to last, checking to see if the SID in the ACE is contained in the process access token (the user SID or one of the user's groups). If so, it checks the desired access rights, against the access mask for the ACE. It keeps doing this until it finds either an ACE that denies/allows access or there are no further ACEs to check (access denied). If a matching ACCESS_ALLOWED ACE_TYPE is found, the system creates a handle that permits the *granted access rights* to the object, and returns the handle to the caller. Otherwise the open request is refused with an ERROR_ACCESS_DENIED error.

Note that all accesses are denied unless specifically allowed. Also once an ACE is found that allows/denies access, the system stops checking ACEs, even if a subsequent ACE contains contradictory access information. For this reason, ACEs that deny access to an object should precede ACEs that allow access. This order can make a significant difference in how quickly a process gains access to the object. A DACL typically denies access to specific users (or groups of users) and then allows access to more general categories of users. The Windows NT File Manager ACL Editor does build DACLs in this 'canonical form'.

There is an important difference between an empty and a nonexistent DACL, specified by the *control* flags of the SD. When a DACL is empty (DACL_PRESENT but it contains no ACEs), no access rights have been explicitly allowed, so access to the object is implicitly denied. On the other hand, when an object has no DACL, no protection is assigned to the object, and any access request is allowed.

Since it is inefficient for the system to perform this check every time a handle is used, on subsequent attempts to use an opened handle, the Object Manager simply compares the granted access rights referenced by the handle with the type of access implied in the API call. However, this means that once a process successfully opens a handle, the granted access rights cannot be revoked by the security system, even if the object's DACL is modified.



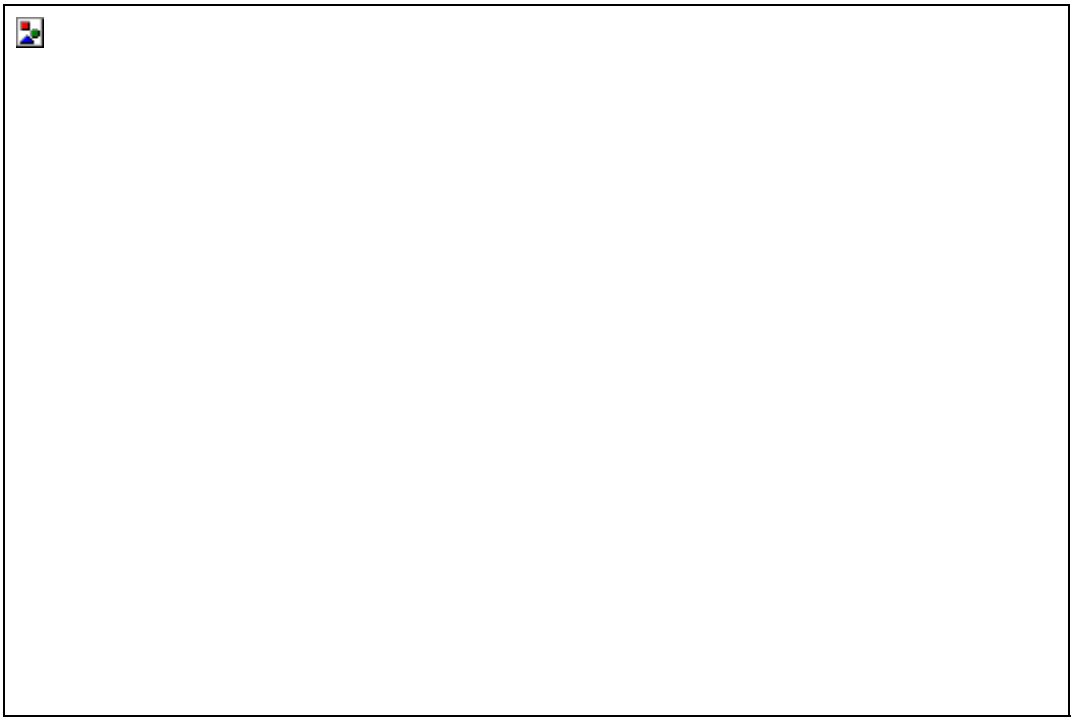
Eric is requesting read, write and execute permission on a file object, and requests access to open a handle to the file object with a desired access mask initialized accordingly. DACL evaluation starts.

The first ACE is evaluated. The SID in the ACE is compared with the user SID in the access token. A match is found and since this is an ACCESS_ALLOWED_TYPE ACE then the read and write bits are cleared in the desired access mask and set in the granted access mask. DACL evaluation carries on.

The second ACE is evaluated. The SID in the ACE is compared with the user SID and then the group SIDS in the access token. A match is found but although this is an ACCESS_ALLOWED_TYPE ACE, Eric has already been granted write permission so DACL evaluation carries on.

The third ACE is evaluated. The SID in the ACE is compared with the user SID and then the group STDs in the access token. A match is found and since this is an ACCESS_ALLOWED_TYPE ACE then the execute bit is cleared in the desired access mask and set in the granted access mask. The desired access mask is cleared and so DACL evaluation carries stops.

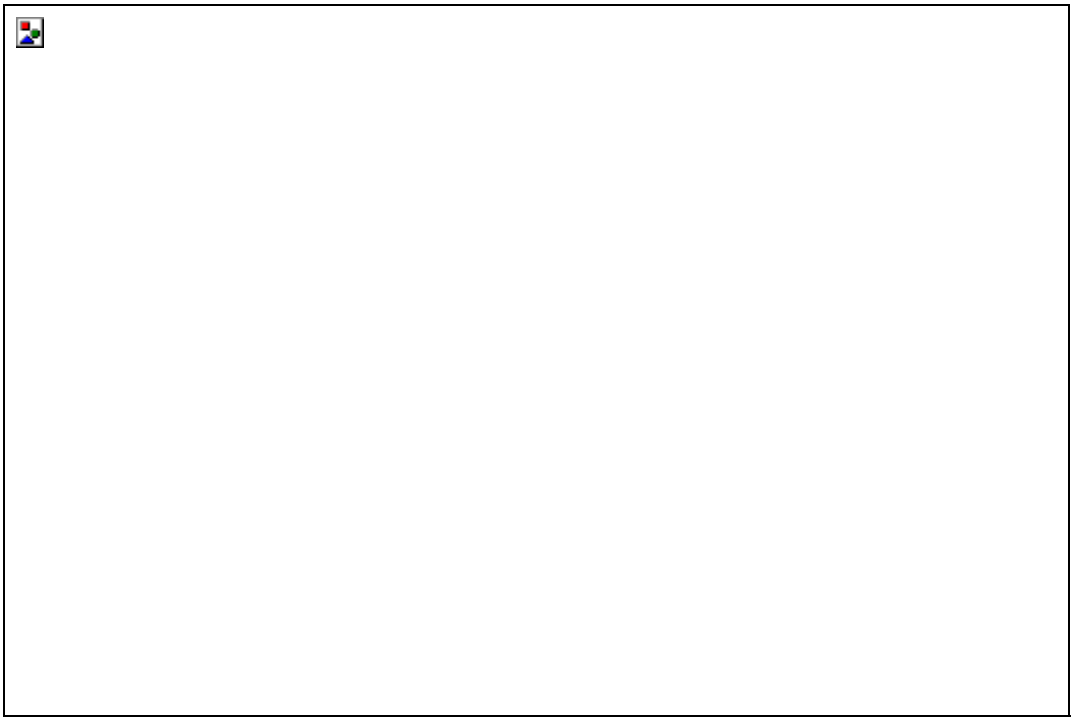
Eric is returned a handle to the file object, which allows him to read from the file, write to the file and execute the file.



Eric is requesting read, write and execute permission on a file object, and requests access to open a handle to the file object with a desired access mask initialized accordingly. DACL evaluation starts.

The first ACE is evaluated. The SID in the ACE is compared with the user SID in the access token. A match is found and since this is an ACCESS_DENIED_TYPE ACE and the write bit is set in the desired access mask and denied by the ACE, DACL evaluation stops.

Eric is returned an ACCESS_DENIED_ERROR error and he does not have a handle that he can use to read from the file, write to the file or execute the file.



A subject is an entity that represents an authenticated user; a process with an access token. There are two kinds of subject; *simple* and *server* subjects. This is to accommodate the client-server model of Windows NT.

A simple subject represents a logged-on user; it is not acting as a server and does not have other subjects as clients. When the simple subject attempts to access objects etc, the access token of the subject is used.

A server subject is a protected subsystem or service process, which has other client processes, which it impersonates. Impersonation is the ability of a process or thread to take on the security profile of another process. When a thread in the server subject is impersonating a client, that **thread** temporarily adopts the clients access token. When the impersonating server thread attempts to access objects etc on behalf of the client, this temporary thread access token is used

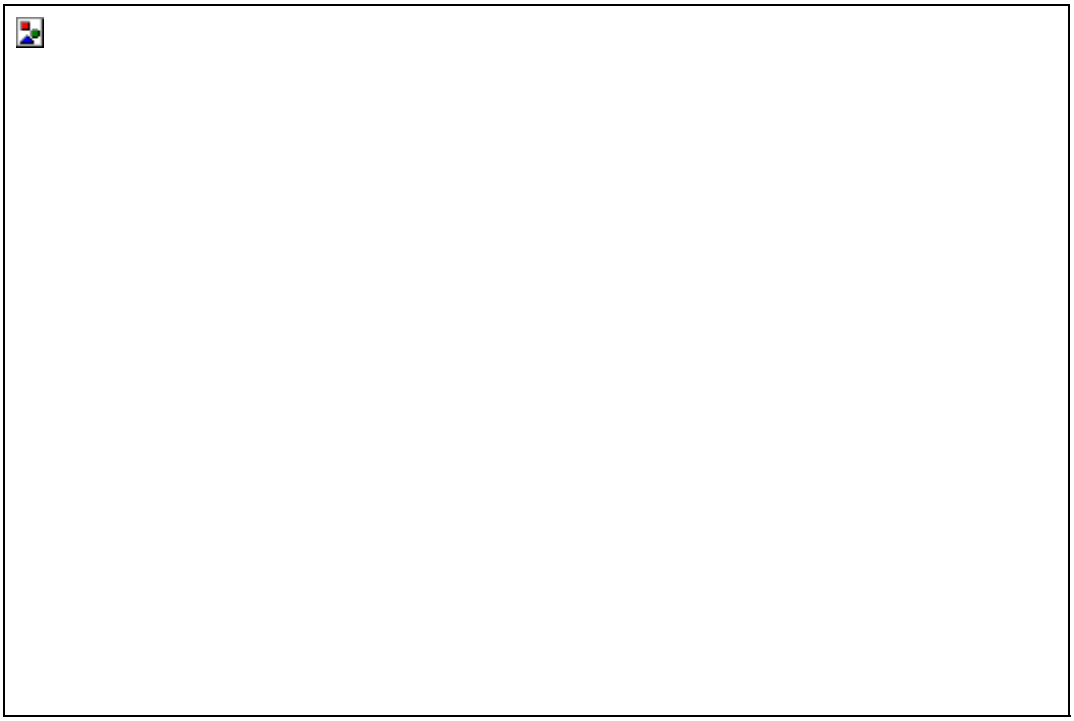
Typically, a server subject impersonates a client process to complete a task involving objects that the client does not normally have access to. If a client does not have an account on the server's domain, the server must impersonate the client to gain access to secured objects. For example, when a client in a DDE conversation requests information from a DDE server, the server may be required to open a file to retrieve that information; this server would impersonate the client when opening the file, enabling the system to verify that the client is allowed access to the information.

A DDE server application can impersonate a client by calling `DdeImpersonateClient()`. When it has finished the task that required the impersonation, it should revert to its own security information by calling `DdeRevertToSelf()`. Similarly, a named-pipe server can use `ImpersonateNamedPipeClient()` and `RevertToSe1f()`. A named pipe server impersonates the security context of the last message read from the pipe.

`DuplicateToken()` allows the creation of an impersonation token that duplicates an existing token. Currently, this supports three enumerated type *impersonation levels*. *SecurityAnonymous* means the server cannot obtain identification information about the client, nor impersonate the client. *SecurityIdentification* allows the server to obtain information about the client, such as security identifiers and privileges, without being able to

impersonate the client. This is useful for servers who export their own objects, because using the client security information; the server is able to make access-validation decisions for itself even though it is unable to use other services using the client's security context. *SecurityImpersonation* allows the server to impersonate the client's security context on its local system. The server cannot impersonate the client on remote systems.

`ImpersonateSelf()` enables a thread to generate a copy of its own access token. This is useful when an application needs to change the security attributes of only a single thread that requires, perhaps, a special privilege.



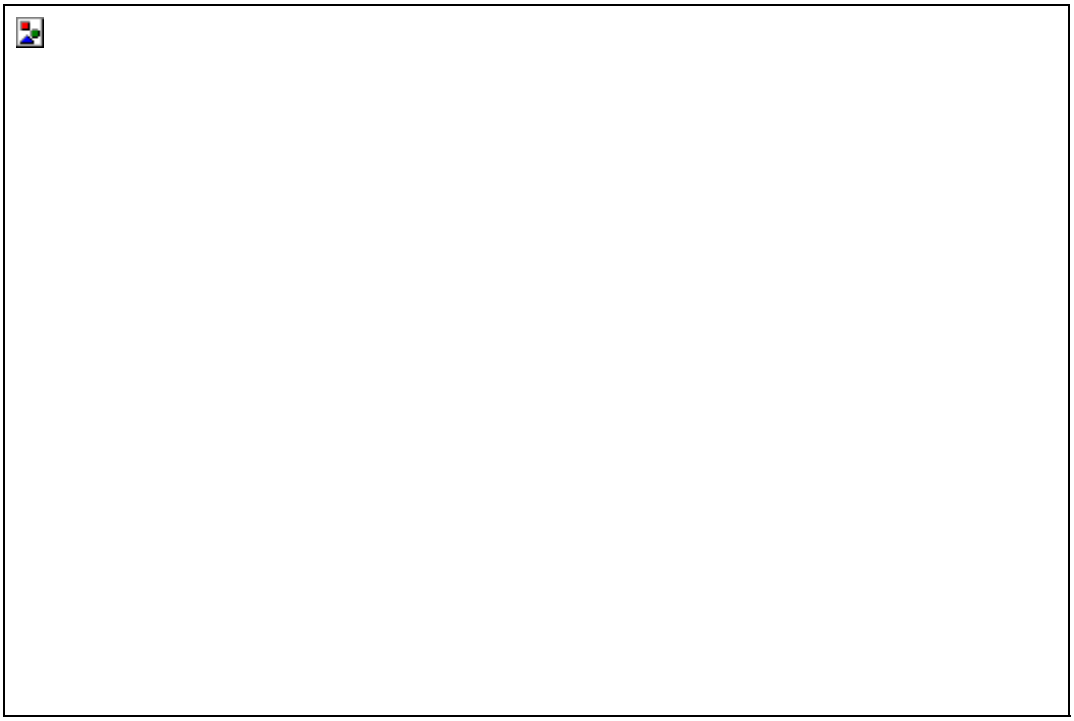
A *Privilege* is a *locally unique identifier* (LUID) used to regulate the use of some system services and system resources; they effectively allow security overrides. Privileges may be identified in 3 ways. A string name, meaningful across systems, called a global program name. For example, *SE_SYSTENTIME_NAME*. A readable name that can be displayed to the user when necessary. For example, “Change the system time”. And a local representation that differs from computer to computer.

Some security-related Win32 API functions, which affect the whole system, require the caller to have certain privileges. An application uses privileges, for example, when it changes the system time or shuts down the system. A Windows NT administrator grants privileges to allow users/groups to have access to system resources they would not normally have control of. For example, the administrator could grant *SE_BACKUP_NAME* privilege to a user, allowing him/her to backup files he/she would not normally have read-access to. Otherwise all files would need to grant explicit access to some account.

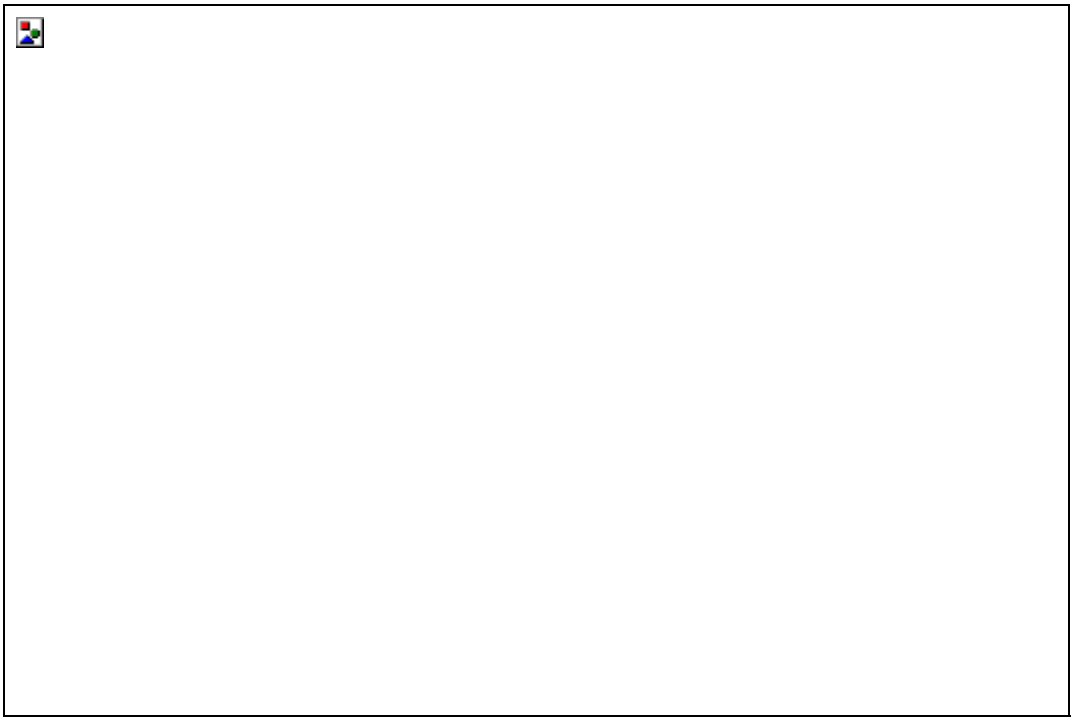
Privileges have two states, enabled and disabled, allowing a user/group to have a privilege but not to use it. This prevents accidental use. Every user gets the sum total of his/her privileges, plus those granted to all of his/her groups. Privileges are very powerful and there is no way to override their effects or restrict their use, as there is with a DACL on an object. They should be used sparingly, and always disabled when not needed.

Most users have no privileges, i.e. the user has privileges that have been disabled and they must be enabled to use them. For example, to set the time on the local computer, a user must first call AdjustTokenPrivileges() to set the *SE_PRIVILEGE_ENABLED* attribute for the *SE_SYSTEMTIME_NAME* privilege.

Future releases of Windows NT will allow user-defined privileges as well as the *well-known privileges* mentioned above.

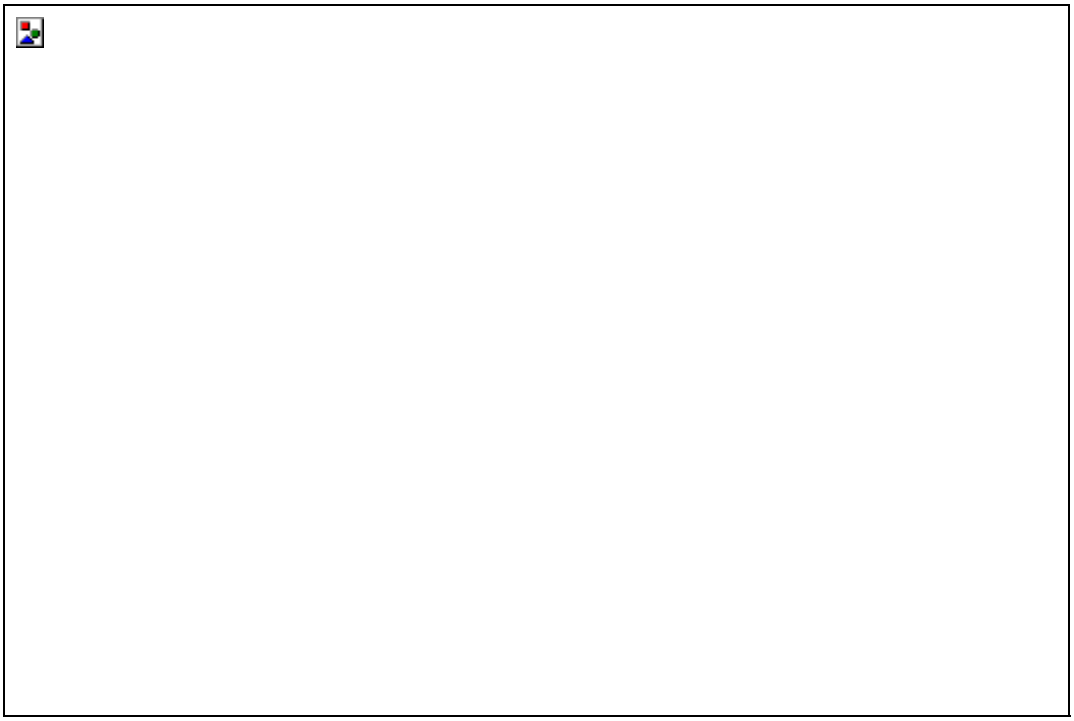


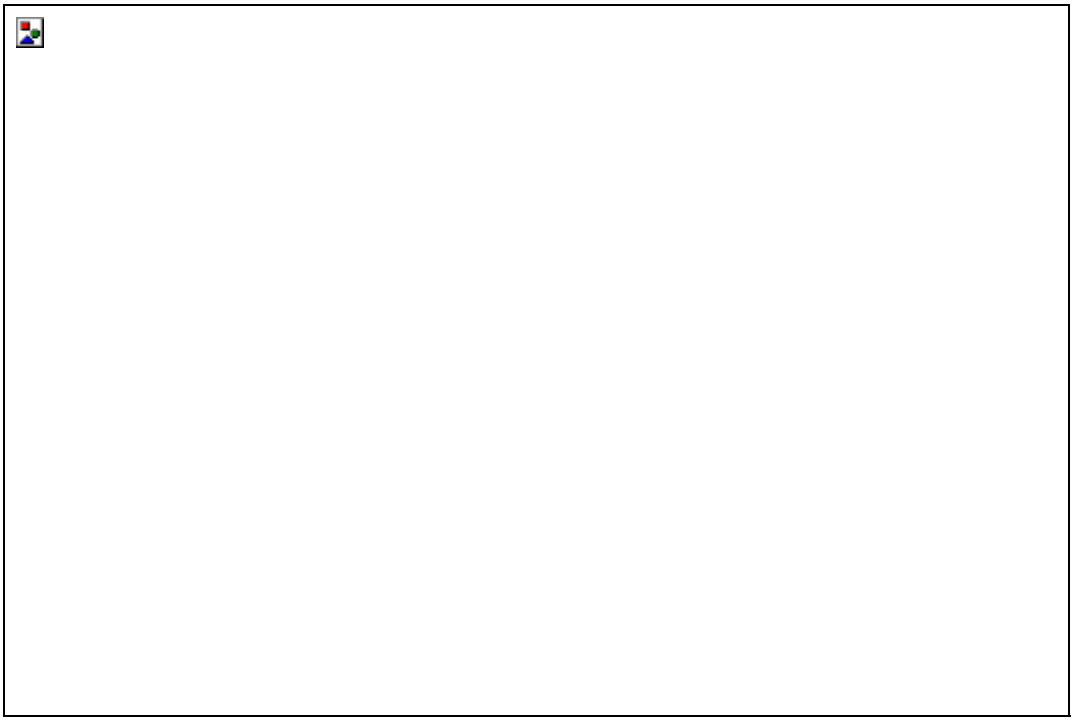
The above code fragment outlines how to use the Win32 API to deny all access to a file. For a full code listing, see the Microsoft Win32 On-line Help under ‘Denying All Access’.



The above code fragment outlines how to find an SID for a particular account and then how to add an access-allowed **ACE** for that account.

For a full code listing, see the Microsoft Win32 On-line Help under '*Allowing Access*'.





Windows NT security is designed in at the lowest level to meet the US government C2 security standards for discretionary access control. Security pervades the whole system and cannot be bypassed, but is discretionary so can be as flexible as required.
The operating system maintains security information for both users, who are required to validate themselves by a logon, and the objects they want to access, such as files, processes, threads etc.
Access control lists, controlled by the object owner, provide a flexible way to provide discretionary access to a variety of different operating system and user-define resources.
Security is controlled at a central point at the time of any object manipulation. Windows NT also supports privileges, which are controlled by the administrator to provide a way to override the normal security mechanisms to handle special problems, like users being able to back-up files that they do not normally have read access to.