BSD Version 4.3 uses a combination of signals (`SIGIO`) to indicate an event on a file descriptor and select a function to determine the ready state of file descriptors. The file descriptors must be set in the `O_ASYNC` mode. This approach works only with terminals and network communication.

Looking Ahead

Chapter 15 completes the book by showing how to secure Windows objects.

Exercises

14–1. Use asynchronous I/O to merge several sorted files into a larger sorted file.

14–2. Does the `FILE_FLAG_NO_BUFFERING` flag improve `cciOV` or `cciEX` performance? Are there any restrictions on file size? Read the MSDN `CreateFile` documentation carefully.

14–3. Experiment with the `cciOV` and `cciEX` record sizes to determine the performance impact. Is the optimal record size machine-independent? What results to you get on Window XP and Windows 7?

14–4. Modify `TimeBeep` (Program 14–3) so that it uses a manual-reset notification timer.

14–5. Modify the named pipe client in Program 11–2, `clientNP`, to use overlapped I/O so that the client can continue operation after sending the request. In this way, it can have several outstanding requests.

14–6. Rewrite the socket server, `serverSK` in Program 12–2, so that it uses I/O completion ports.

14–7. Rewrite either `serverSK` or `serverNP` so that the number of ready worker threads is limited by a semaphore. Experiment with a large thread pool to determine the effectiveness of this alternative.

14–8. Use `JobShell` (Program 6–3, the job management program) to bring up a large number of clients and compare the responsiveness of `serverNP` and `serverCP`. Networked clients can provide additional load. Determine an optimal range for the number of active threads.

14–9. Modify `cciMT` to use an NT6 thread pool rather than thread management. What is the performance impact? Compare your results with those in Appendix C.

14–10. Modify `cciMT` to use overlapped read/write calls with the event wait immediately following the read/write. This should allow you to cancel I/O operations with a user APC; try to do so. Also, does this change affect performance?

14–11. Modify `serverCP` so that there is no limit on the number of clients. Use `PIPE_UNLIMITED_INSTANCES` for the `CreateNamedPipe` `nMaxInstances` value. You will need to replace the array of `CP_KEY` structures with dynamically allocated structures.

14–12. Review `serverCP`'s error and disconnect processing to be sure all situations are covered. Fix any deficiencies.

CHAPTER

15 Securing Windows Objects

Windows supports a comprehensive security model that prevents unauthorized access to objects such as files, processes, and file mappings. Nearly all sharable objects can be protected, and the programmer has a fine granularity of control over access rights. Windows has Common Criteria Certification at Evaluation Assurance Level 4 (EAL-4), an internationally recognized criteria.

Security is a large subject that cannot be covered completely in a single chapter. Therefore, this chapter concentrates on the immediate problem of showing how to use the Windows security API to protect objects from unauthorized access. While access control is only a subset of Windows security functionality, it is of direct concern to those who need to add security features to their programs. The initial example, Program 15–1, shows how to emulate UNIX file permissions with NT file system (NTFS) files, and a second example applies security to named pipes. The same principles can then be used to secure other objects. The bibliography lists several resources you can consult for additional security information.

Security Attributes

This chapter explores Windows access control by proceeding from the top down to show how to construct an object's security. Following an overview, the Windows functions are described in detail before proceeding to the examples. In the case of files, it is also possible to use Windows Explorer to examine and manage some file security attributes.

Nearly any object created with a `Create` system call has a security attributes parameter. Therefore, programs can secure files, processes, threads, events, semaphores, named pipes, and so on. The first step is to include a `SECURITY_ATTRIBUTES` structure in the `Create` call. Until now, our programs have always used a `NULL` pointer in `Create` calls or have used `SECURITY_ATTRIBUTES` simply to create inheritable handles (Chapter 6). In order to implement security, the important element in the `SECURITY_ATTRIBUTES` structure is `lpSecurityDescriptor`, the pointer to a *security descriptor,* which describes the object's owner and determines which users are allowed or denied various rights.

An individual process is identified by its *access token,* which specifies the owning user and group membership. When a process attempts to access an object, the Windows kernel can determine the process's identity using the token and can then decide from the information in the security descriptor whether or not the process has the required rights to access the object.

Chapter 6 introduced the `SECURITY_ATTRIBUTES` structure; for review, here is the complete structure definition:

```
typedef struct _SECURITY_ATTRIBUTES {
   DWORD nLength;
   LPVOID lpSecurityDescriptor;
   BOOL bInheritHandle;
} SECURITY_ATTRIBUTES, *PSECURITY_ATTRIBUTES;
```

Set `nLength` to `sizeof(SECURITY_ATTRIBUTES)`. `bInheritHandle` indicates whether or not the handle is inheritable by other processes.

Security Overview: The Security Descriptor

Analyzing the security descriptor gives a good overview of essential Windows security elements. This section mentions the various elements and the names of the functions that manage them, starting with security descriptor structure.

A security descriptor is initialized with the function `InitializeSecurityDescriptor`, and it contains the following:

- The owner security identifier (SID) (described in the next section, which deals with the object's owner)
- The group SID
- A discretionary access control list (DACL)—a list of entries explicitly granting and denying access rights. The term "ACL" without the "D" prefix will refer to DACLs in our discussion.
- A system ACL (SACL), sometimes called an "audit access ACL," controls audit message generation when programs access securable objects; you need to have system administrator rights to set the SACL.

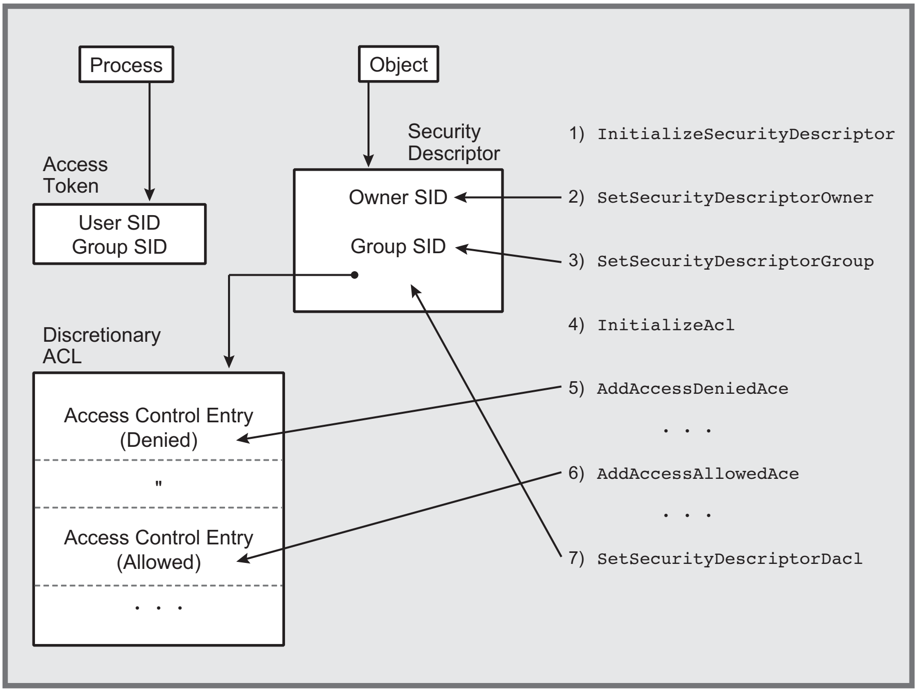

Figure 15–1 Constructing a Security Descriptor

`SetSecurityDescriptorOwner` and `SetSecurityDescriptorGroup` associate SIDs with security descriptors, as described in the upcoming "Security Identifiers" section.

ACLs are initialized using the `InitializeAcl` function and are then associated with a security descriptor using `SetSecurityDescriptorDacl` or `SetSecurityDescriptorSacl`.

Figure 15–1 shows the security descriptor and its components.

Access Control Lists

Each ACL is a set (list) of access control entries (ACEs). There are two types of ACEs: one for access allowed and one for access denied.

You first initialize an ACL with `InitializeAcl` and then add ACEs. Each ACE contains a SID and an *access mask,* which specifies rights to be granted or denied to the user or group specified by the SID. `FILE_GENERIC_READ` and `DELETE` are typical file access rights.

The two functions used to add ACEs to discretionary ACLs are `AddAccessAllowedAce` and `AddAccessDeniedAce`. `AddAuditAccessAce` is for adding to an SACL. Finally, remove ACEs with `DeleteAce` and retrieve them with `GetAce`.

Using Windows Object Security

There are numerous details to fill in, but Figure 15–1 shows the basic structure. Notice that each process also has SIDs (in an access token), which the kernel uses to determine whether access is allowed. The user's access token may also give the owner certain *privileges* (the ability to perform system operations such as system shutdown and to access system resources). These user and group privileges are set when the administrator creates the account.

The kernel scans the ACL for access rights for the user, based on the user's ID and group. The first entry that specifically grants or denies the requested service is decisive. The order in which ACEs are entered into an ACL is therefore important. Frequently, access-denied ACEs come first so that a user who is specifically denied access will not gain access by virtue of membership in a group that does have such access. In Program 15–1, however, it is essential to mix allowed and denied ACEs to obtain the desired semantics.

Object Rights and Object Access

An object, such as a file, gets its security descriptor at creation time, although the program can change the security descriptor at a later time.

A process requests access to the object when it asks for a handle using, for example, a call to `CreateFile`. The handle request contains the desired access, such as `FILE_GENERIC_READ`, in one of the parameters. If the security descriptor grants access to the process, the request succeeds. Different handles to the same object may have different access rights. The access flag values are the same for both allowing and denying rights when creating ACLs.

Standard UNIX provides a simpler security model. It is limited to files and based on file permissions. The example programs in this chapter emulate the UNIX permissions.

Security Descriptor Initialization

The first step is to initialize the security descriptor using the `InitializeSecurityDescriptor` function. Set the `pSecurityDescriptor` parameter to the address of a valid `SECURITY_DESCRIPTOR` structure. These structures are opaque and are managed with specific functions.

Security descriptors are classified as either *absolute* or *self-relative*. This distinction is ignored for now but is explained near the end of the chapter.

```
BOOL InitializeSecurityDescriptor (
   PSECURITY_DESCRIPTOR pSecurityDescriptor,
   DWORD dwRevision)
```

dwRevision is set to the constant SECURITY_DESCRIPTOR_REVISION.

Security Descriptor Control Flags

Flags within the Control structure of the security descriptor, the SECURITY_DESCRIPTOR_CONTROL flags, control the meaning assigned to the security descriptor. Several of these flags are set or reset by the upcoming functions and will be mentioned as needed. GetSecurityDescriptorControl and SetSecurityDescriptorControl access these flags, but the examples do not use the flags directly.

Security Identifiers

Windows uses SIDs to identify users and groups. The program can look up a SID from the account name, which can be a user, group, domain, and so on. The account can be on a remote system. The first step is to determine the SID from an account name.

```
BOOL LookupAccountName (
   LPCTSTR lpSystemName,
   LPCTSTR lpAccountName,
   PSID Sid,
   LPDWORD cbSid,
   LPTSTR ReferencedDomainName,
   LPDWORD cbReferencedDomainName,
   PSID_NAME_USE peUse)
```

Parameters

lpSystemName and lpAccountName point to the system and account names. Frequently, lpSystemName is NULL to indicate the local system.

Sid is the returned information, which is of size *cbSid. The function will fail, returning the required size, if the buffer is not large enough.

ReferencedDomainName is a string of length *cbReferencedDomainName characters. The length parameter should be initialized to the buffer size (use the usual techniques to process failures). The return value shows the domain where the name is found. The account name Administrators will return BUILTIN, whereas a user account name will return that same user name.

peUse points to a SID_NAME_USE (enumerated type) variable and can be tested for values such as SidTypeWellKnownGroup, SidTypeUser, SidTypeGroup, and so on.

Getting the Account and User Names

Given a SID, you reverse the process and obtain the account name using LookupAccountSid. Specify the SID and get the name in return. The account name can be any name available to the process. Some names, such as Everyone, are well known.

```
BOOL LookupAccountSid (
   LPCTSTR lpSystemName,
   PSID lpSid,
   LPTSTR lpAccountName,
   LPDWORD cchName,
   LPTSTR lpReferencedDomainName,
   LPDWORD cchReferencedDomainName,
   PSID_NAME_USE peUse)
```

Obtain the process's user account name (the logged-in user) with the GetUserName function.

```
BOOL GetUserName (
   LPTSTR lpBuffer,
   LPDWORD lpnSize)
```

The user name and length are returned in the conventional manner.

Create and manage SIDs using functions such as InitializeSid and AllocateAndInitializeSid. The examples confine themselves, however, to SIDs obtained from account names.

Once SIDs are known, they can be entered into an initialized security descriptor.

```
BOOL SetSecurityDescriptorOwner (
   PSECURITY_DESCRIPTOR pSecurityDescriptor,
   PSID pOwner,
   BOOL bOwnerDefaulted)
```

```
BOOL SetSecurityDescriptorGroup (
   PSECURITY_DESCRIPTOR pSecurityDescriptor,
   PSID pGroup,
   BOOL bGroupDefaulted)
```

`pSecurityDescriptor` points to the appropriate security descriptor, and `pOwner` (or `pGroup`) is the address of the owner's (group's) SID. As always in such situations, assure that these SIDs were not prematurely freed.

`bOwnerDefaulted` (or `bGroupDefaulted`) indicates, if `TRUE`, that a default mechanism is used to derive the owner (or primary group) information. The `SE_OWNER_DEFAULTED` and `SE_GROUP_DEFAULTED` flags within the `SECURITY_DESCRIPTOR_CONTROL` structure are set according to these two parameters.

The similar functions `GetSecurityDescriptorOwner` and `GetSecurityDescriptorGroup` return the SID (either owner or group) from a security descriptor.

Managing ACLs

This section shows how to manage ACLs, how to associate an ACL with a security descriptor, and how to add ACEs. Figure 15–1 shows the relationships between these objects and functions.

The first step is to initialize an ACL structure. The ACL should not be accessed directly, so its internal structure is not relevant. The program must, however, provide a buffer to serve as the ACL; the functions manage the contents.

```
BOOL InitializeAcl (
   PACL pAcl,
   DWORD cbAcl,
   DWORD dwAclRevision)
```

`pAcl` is the address of a programmer-supplied buffer of `cbAcl` bytes. Subsequent discussion and Program 15–4 will show how to determine the ACL size, but 1KB is more than adequate for most purposes. `dwAclRevision` should be `ACL_REVISION`.

Next, add the ACEs in the order desired with the `AddAccessAllowedAce` and `AddAccessDeniedAce` functions.

```
BOOL AddAccessAllowedAce (
   PACL pAcl,
   DWORD dwAclRevision
   DWORD dwAccessMask,
   PSID pSid)

BOOL AddAccessDeniedAce (
   PACL pAcl,
   DWORD dwAclRevision,
   DWORD dwAccessMask,
   PSID pSid)
```

`pAcl` points to the same ACL structure initialized with `InitializeAcl`, and `dwAclRevision` is `ACL_REVISION`. `pSid` points to a SID, such as one that would be obtained from `LookupAccountName`.

The access mask (`dwAccessMask`) determines the rights to be granted or denied to the user or group specified by the SID. The predefined mask values will vary by the object type.

The final step is to associate an ACL with the security descriptor. In the case of the discretionary ACL, use the `SetSecurityDescriptorDacl` function.

```
BOOL SetSecurityDescriptorDacl (
   PSECURITY_DESCRIPTOR pSecurityDescriptor,
   BOOL bDaclPresent,
   PACL pAcl,
   BOOL fDaclDefaulted)
```

`bDaclPresent`, if `TRUE`, indicates that there is an ACL in the `pAcl` structure. If `FALSE`, `pAcl` and `fDaclDefaulted`, the next two parameters, are ignored. The `SECURITY_DESCRIPTOR_CONTROL`'s `SE_DACL_PRESENT` flag is also set to this parameter's value.

The final flag is `fDaclDefaulted`. `FALSE` indicates an ACL generated by the programmer. `TRUE` indicates that the ACL was obtained by a default mechanism, such as inheritance. The `SE_DACL_DEFAULTED` flag in the `SECURITY_DESCRIPTOR-_CONTROL` is set to this parameter value.

Other functions delete ACEs and read ACEs from an ACL; we discuss them later in this chapter. It is now time for an example.

Example: UNIX-Style Permission for NTFS Files

UNIX file permissions provide a convenient way to illustrate Windows security, even though Windows security is much more general than standard UNIX security.

First, however, here is a very quick review of UNIX file permissions (directories are treated slightly differently).

- Every file has an owning user and group.
- Every file has 9 permission bits, which are specified as 3 octal (base 8) digits.
- The 3 bits in each octal digit grant, or deny, read (high-order bit), write, and execute (low-order bit) permission. Read, write, and execute permissions are displayed as `r`, `w`, and `x` respectively. Execute rights are meaningful for `.exe` and `.bat` files but not for `.txt` files.
- The 3 octal digits, from left to right, represent rights given to the owner, the group, and to everyone else.
- Thus, if you set the permissions to `640`, the permissions will be displayed as `rw_r______`. The file owner can read and write the file, group members can read it, and everyone else has no access.

The implementation creates nine ACEs to grant or deny read, write, and execute permissions to the owner, group, and everyone. There are two commands.

1. `chmodW` sets the permissions and is modeled after the UNIX `chmod` command. The implementation has been enhanced to create the specified file if it does not already exist and to allow the user to specify the group name.
2. `lsFP` displays the permissions along with other file information and is an extension of the `lsW` command (Program 3–2). When the long listing is requested, the command displays the owning user and an interpretation of the existing ACLs, which may have been set by `chmodW`.

Programs 15–1 and 15–2 show the implementation for these two commands. Programs 15–3, 15–4, and 15–5 show three supporting functions:

1. `InitializeUnixSA`, which creates a valid security attributes structure corresponding to a set of UNIX permissions. This function is general enough that it can be used with objects other than files, such as processes (Chapter 6), named pipes (Chapter 11), and synchronization objects (Chapter 8).
2. `ReadFilePermissions`.
3. `ChangeFilePermissions`.

Note: The separate `DeniedAceMasks` array assures that `SYNCHRONIZE` rights are never denied because the `SYNCHRONIZE` flag is set in all three of the macros, `FILE_GENERIC_READ`, `FILE_GENERIC_WRITE`, and `FILE_GENERIC_EXECUTE`, which are combinations of several flags (see the include file, `winnt.h`). The full program in the *Examples* file provides additional explanation.

The programs that follow are simplifications of the programs from the *Examples* file. For example, the full program checks to see if there is a group name on the command line; here, the name is assumed. Also, there are command line flags to create a file that does not exist and to suppress the warning message if the change fails.

Program 15–1 `chmodW`: Change File Permissions

```
/* Chapter 15-1. Windows chmodW command */
/* chmodW [options] mode file [groupName]
   Update access rights of the named file. */
/* This program illustrates:
   1.  Setting the file security attributes.
   2.  Changing a security descriptor. */
```