```
#include "Everything.h"

int _tmain (int argc, LPTSTR argv[])
{
   HANDLE hFile, heap = NULL;
   BOOL force, createNew, change, exists;
   DWORD mode, userCount = ACCT_NAME_SIZE;
   TCHAR userName[ACCT_NAME_SIZE];
   int fileIndex, groupIndex, modeIndex;
      /* Array of rights settings in "UNIX order". */
   DWORD allowedAceMasks[] =
      {FILE_GENERIC_READ, FILE_GENERIC_WRITE, FILE_GENERIC_EXECUTE};
   DWORD deniedAceMasks[] =
      {  FILE_GENERIC_READ & ~SYNCHRONIZE,
         FILE_GENERIC_WRITE & ~SYNCHRONIZE,
         FILE_GENERIC_EXECUTE & ~SYNCHRONIZE};
   LPSECURITY_ATTRIBUTES pSa;
   LPCTSTR groupName = NULL;

   modeIndex = Options (argc, argv, _T ("fc"),
                     &force, &createNew, NULL);
   groupIndex = modeIndex + 2;
   fileIndex = modeIndex + 1;
   /* Mode is base 8, convert to decimal */
   mode = _tcstoul(argv[modeIndex], NULL, 8);
   exists = (_taccess (argv[fileIndex], 0) == 0);
   if (!exists && createNew) {
      GetUserName (userName, &userCount);
      pSa = InitializeUnixSA (mode, userName, groupName,
            allowedAceMasks, deniedAceMasks, &heap);
      hFile = CreateFile (argv[fileIndex], 0,
            0, pSa, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
      CloseHandle (hFile);
   }
   else if (exists) {
         /* File exists; change permissions. */
      change = ChangeFilePermissions (mode, argv[fileIndex],
               allowedAceMasks,deniedAceMasks);
   }

   DestroyUnixSA (pSa, heap);
   return 0;
}
```

Program 15–2 shows the relevant part of `lsFP`—namely, the `ProcessItem` function. Other parts of the program are similar to Chapter 3's `lsW` Program 3–2.

Program 15–2 `lsFP`: List File Permissions

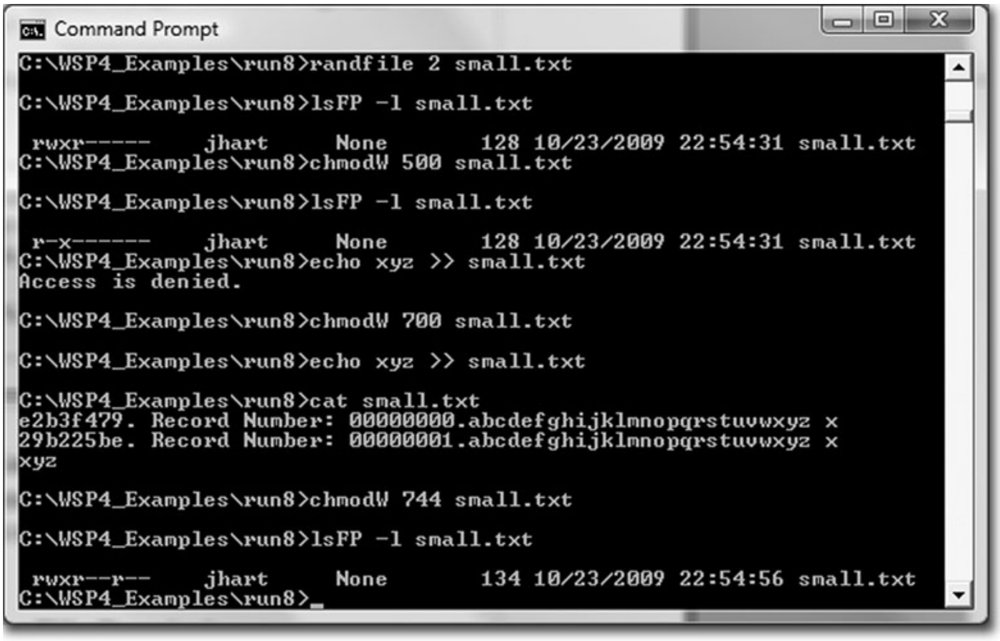
```
static BOOL ProcessItem (LPWIN32_FIND_DATA pFileData,
      DWORD numberFlags, LPBOOL flags)

/* List attributes, with file permissions and owner. */
{
   DWORD fileType = FileType(pFileData), Mode, i;
   BOOL dashL = flags[1];
   TCHAR groupName[ACCT_NAME_SIZE], userName[ACCT_NAME_SIZE];
   SYSTEMTIME timeLastWrite;
   TCHAR  permissionString[] = _T("---------");
   const TCHAR RWX[] = {'r','w','x'}, FileTypeChar[] = {' ','d'};

   if (fileType != TYPE_FILE && fileType != TYPE_DIR)
      return FALSE;
   _tprintf (_T ("\n"));

   if (dashL) {
      Mode = ReadFilePermissions (pFileData->cFileName,
            userName, groupName);
      if (Mode == 0xFFFFFFFF)
         Mode = 0;
      for (i = 0; i < 9; i++) {
         if ( (Mode / (1 << (8 - i)) & 0x1) )
             permissionString[i] = RWX[i % 3];
      }
      _tprintf (_T ("%c%s %8.7s %8.7s%10d"),
         FileTypeChar[fileType-1], permissionString, userName,
            groupName,pFileData->nFileSizeLow);
      FileTimeToSystemTime (&(pFileData->ftLastWriteTime),
                          &timeLastWrite);
      _tprintf (_T (" %02d/%02d/%04d %02d:%02d:%02d"),
            timeLastWrite.wMonth, timeLastWrite.wDay,
            timeLastWrite.wYear, timeLastWrite.wHour,
            timeLastWrite.wMinute, timeLastWrite.wSecond);
   }
   _tprintf (_T (" %s"), pFileData->cFileName);
   return TRUE;
}
```

The next step is to show the supporting function implementations. However, Run 15–2 first shows the two new commands in operation. First, a new file is created, and its permissions are seen to be 700 (the owner can read, write, and execute the file; others have no rights). Next, the owner's write permission is removed, and an attempt to write to the file is denied. Once the write permissions are restored, the file write succeeds, and the file listing (`cat` command) shows that the new text is at the end of the file. Finally, others are given read permission.



```
C:\WSP4_Examples\run8>randfile 2 small.txt

C:\WSP4_Examples\run8>lsFP -l small.txt

 rwxr-----    jhart     None       128 10/23/2009 22:54:31 small.txt
C:\WSP4_Examples\run8>chmodW 500 small.txt

C:\WSP4_Examples\run8>lsFP -l small.txt

 r-x------    jhart     None       128 10/23/2009 22:54:31 small.txt
C:\WSP4_Examples\run8>echo xyz >> small.txt
Access is denied.

C:\WSP4_Examples\run8>chmodW 700 small.txt

C:\WSP4_Examples\run8>echo xyz >> small.txt

C:\WSP4_Examples\run8>cat small.txt
e2b3f479. Record Number: 00000000.abcdefghijklmnopqrstuvwxyz x
29b225be. Record Number: 00000001.abcdefghijklmnopqrstuvwxyz x
xyz

C:\WSP4_Examples\run8>chmodW 744 small.txt

C:\WSP4_Examples\run8>lsFP -l small.txt

 rwxr--r--    jhart     None       134 10/23/2009 22:54:56 small.txt
C:\WSP4_Examples\run8>_
```

Run 15–2 `chmodW, lsFP:` UNIX-like File Permissions

Example: Initializing Security Attributes

Program 15–3 shows the utility function `InitializeUnixSA`, which creates a security attributes structure containing an ACL with ACEs that emulate UNIX file permissions. There are nine ACEs granting or denying read, write, and execute permissions for the owner, the group, and everyone else. The actual array of three rights (read, write, and execute for files) can vary according to the object type being secured. This structure is not a local variable in the function but must be allocated and initialized and then returned to the calling program; notice the ACE mask arrays in Program 15–1.

Two aspects of this program are interesting and could be modified (see the exercises).

- The function creates a heap and allocates memory from the heap. This greatly simplifies destroying the SA (`DestroyUnixSA` is at the end). The heap is returned from the function; alternatively, you could create an opaque structure containing the heap and the SA.

- The SDs in the SA are “absolute” rather than self-relative; a later section talks about this some more.

Program 15–3 `InitUnSA`: Initializing Security Attributes

```
/* Set UNIX-style permissions as ACEs in a
   SECURITY_ATTRIBUTES structure. */

#include "Everything.h"

#define ACL_SIZE 1024
#define SID_SIZE SECURITY_MAX_SID_SIZE
#define DOM_SIZE LUSIZE

static VOID FindGroup (DWORD, LPTSTR, DWORD);

LPSECURITY_ATTRIBUTES InitializeUnixSA (DWORD unixPerms,
      LPTSTR usrName, LPTSTR grpName, LPDWORD allowedAceMasks,
      LPDWORD deniedAceMasks, LPHANDLE pSaHeap)

/* Allocate a structure and set the UNIX style permissions
   as specified in unixPerms, which is 9-bits
   (low-order end) giving the required [r,w,x] settings
   for [User,Group,Other] in the familiar UNIX form.
   Return a pointer to a security attributes structure */

{
   HANDLE saHeap = HeapCreate (HEAP_GENERATE_EXCEPTIONS, 0, 0);
   /*  Several memory allocations are necessary to build the SA
      and they are all constructed in this heap.
      This memory MUST be available to the calling program and must
      not be allocated on the stack of this function.*/

   LPSECURITY_ATTRIBUTES pSA = NULL;
   PSECURITY_DESCRIPTOR pSD = NULL;
   PACL pAcl = NULL;
   BOOL success, ok = TRUE;
   DWORD iBit, iSid;

   /* Various tables of User, Group, and Everyone Names, SIDs,
      etc. for use first in LookupAccountName and SID creation. */

   LPTSTR groupNames[3] = {EMPTY, EMPTY, _T ("Everyone")};
   PSID pSidTable[3] = {NULL, NULL, NULL};
   SID_NAME_USE sNamUse[] =
      {SidTypeUser, SidTypeGroup, SidTypeWellKnownGroup};
   TCHAR refDomain[3][DOM_SIZE];
   DWORD refDomainCount[3] = {DOM_SIZE, DOM_SIZE, DOM_SIZE};
   DWORD sidCount[3] = {SID_SIZE, SID_SIZE, SID_SIZE};
```

```
__try {
   /* This is in a try-except block so as to
      free resources in case of any subsequent failure. */

   pSA = HeapAlloc (saHeap, 0, sizeof (SECURITY_ATTRIBUTES));
   pSA->nLength = sizeof (SECURITY_ATTRIBUTES);
   pSA->bInheritHandle = FALSE; /* Programmer can set this later. */

   pSD = HeapAlloc (saHeap, 0, sizeof (SECURITY_DESCRIPTOR));
   pSA->lpSecurityDescriptor = pSD;
   /* Other function calls are tested, but just this test is shown */
   if (!InitializeSecurityDescriptor (pSD,
                   SECURITY_DESCRIPTOR_REVISION))
      ReportException (_T ("I.S.D. Error"), 21);

   /* Set up the table names for the user and group.
      Then get a SID for User, Group, and Everyone. */

   groupNames[0] = usrName;
   if (grpName == NULL || _tcslen(grpName) == 0) {
      /*  No group name specified. Get the user's primary group. */
      /*  Allocate a buffer for the group name */
      groupNames[1] = HeapAlloc (saHeap, 0, ACCT_NAME_SIZE);
      FindGroup (2, groupNames[1], ACCT_NAME_SIZE);
   } else groupNames[1] = grpName;

   /* Look up the three names, creating the SIDs. */
   for (iSid = 0; iSid < 3; iSid++) {
      pSidTable[iSid] = HeapAlloc (saHeap, 0, SID_SIZE);
      LookupAccountName (NULL, groupNames[iSid],
            pSidTable[iSid], &sidCount[iSid],
            refDomain[iSid], &refDomainCount[iSid], &sNamUse[iSid]);
   }

   /* Set the security descriptor owner & group SIDs. */
   SetSecurityDescriptorOwner (pSD, pSidTable[0], FALSE);
   SetSecurityDescriptorGroup (pSD, pSidTable[1], FALSE);

   /* Allocate a structure for the ACL. */
   pAcl = HeapAlloc (saHeap, 0, ACL_SIZE);

   /* Initialize an ACL. */
   InitializeAcl (pAcl, ACL_SIZE, ACL_REVISION);

   /* Add the ACEs. Scan permission bits, adding allowed ACE when
      the bit is set and a denied ACE when the bit is reset. */
   for (iBit = 0; iBit < 9; iBit++) {
      if ((unixPerms >> (8 - iBit) & 0x1) != 0
               && allowedAceMasks[iBit % 3] != 0)
            AddAccessAllowedAce (pAcl, ACL_REVISION,
                  allowedAceMasks[iBit % 3], pSidTable[iBit / 3]);
         else if (deniedAceMasks[iBit % 3] != 0)
            AddAccessDeniedAce (pAcl, ACL_REVISION,
                  deniedAceMasks[iBit % 3], pSidTable[iBit / 3]);
      }

      /* ACL is now complete. Associate it with security descriptor. */
      SetSecurityDescriptorDacl (pSD, TRUE, pAcl, FALSE);
      IsValidSecurityDescriptor (pSD);
   }  /* End of __try-except block. */

   __except ((GetExceptionCode() != STATUS_NO_MEMORY) ?
            EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
      /*  An exception occurred and was reported. All memory allocated to
         create the security descriptor and attributes is freed with a
         single HeapDestroy so that the individual elements do not need
         to be deallocated. */
      if (pSaHeap != NULL)
         HeapDestroy (pSaHeap);
      pSA = NULL;
   }
      return pSA;
   }
```

Comments on Program 15–3

Program 15–3 may have a straightforward structure, but its operation is hardly simple. Furthermore, it illustrates several points about Windows security that need review.

- Several memory allocations are required to hold information such as the SIDs. They are created in a dedicated heap, which the calling program eventually destroys. The advantage is that it's simple to free the allocated memory (there are seven allocations) with a single `HeapDestroy` call.
- The security attribute structure in this example is for files, but it is also used with other objects such as named pipes (Chapter 11). Program 15–4 shows how to integrate the security attributes with a file.
- To emulate UNIX behavior, the ACE entry order is critical. Notice that access-denied and access-allowed ACEs are added to the ACL as the permission bits are processed from left (`Owner/Read`) to right (`Everyone/Execute`). In this way, permission bits of, say, `460` (in octal) will deny write access to the user even though the user may be in the group.

- The ACEs' rights are access values, such as `FILE_GENERIC_READ` and `FILE_GENERIC_WRITE`, which are similar to the flags used with `CreateFile`. The calling program (Program 15–1 in this case) specifies the rights that are appropriate for the object.
- The defined constant `ACL_SIZE` is large enough to contain the nine ACEs. After Program 15–5, it will be apparent how to determine the required size.
- The function uses three SIDs, one each for `User`, `Group`, and `Everyone`. Three different techniques are employed to get the name to use as an argument to `LookupAccountName`. The user name comes from `GetUserName`, or get the user SID from the current token without getting the user name (also avoiding the problem of getting an impersonating name; this is Exercise 15–5). The name for everyone is `Everyone` in a `SidTypeWellKnownGroup`. The group name is a command line argument and is looked up as a `SidTypeGroup`. Finding the groups that the current user belongs to requires some knowledge of process token, and solving this problem is Exercise 15–12. Incidentally, finding the groups of an arbitrary user is fairly complex.
- The version of the program in the *Examples* file, but not the one shown here, is fastidious about error checking. It even goes to the effort to validate the generated structures using the self-explanatory `IsValidSecurityDescriptor`, `IsValidSid`, and `IsValidAcl` functions. This error testing proved to be helpful during debugging.

Reading and Changing Security Descriptors

Now that a security descriptor is associated with a file, the next step is to determine the security of an existing file and, in turn, change it. The following functions get and set file security in terms of security descriptors.

```
BOOL GetFileSecurity (
   LPCTSTR lpFileName,
   SECURITY_INFORMATION secInfo,
   PSECURITY_DESCRIPTOR pSecurityDescriptor,
   DWORD cbSd,
   LPDWORD lpcbLengthNeeded)

BOOL SetFileSecurity (
   LPCTSTR lpFileName,
   SECURITY_INFORMATION secInfo,
   PSECURITY_DESCRIPTOR pSecurityDescriptor)
```

secInfo is an enumerated type that takes on values such as OWNER-_SECURITY_INFORMATION, GROUP_SECURITY_INFORMATION, DACL_SECURITY-_INFORMATION, and SACL_SECURITY_INFORMATION to indicate what part of the security descriptor to get or set. Combine these values with the bit-wise "or" operator.

To figure out the size of the return buffer for GetFileSecurity, the best strategy is to call the function twice. The first call simply uses 0 as the cbSd value. After allocating a buffer, call the function a second time. Program 15–4 operates this way.

Needless to say, the correct file permissions are required in order to carry out these operations. For example, it is necessary to have WRITE_DAC permission or to be the object's owner to succeed with SetFileSecurity.

The functions GetSecurityDescriptorOwner and GetSecurity-DescriptorGroup can extract the SIDs from the security descriptor obtained with GetFileSecurity. Obtain the ACL with the GetSecurityDescriptor-Dacl function.

```
BOOL GetSecurityDescriptorDacl (
   PSECURITY_DESCRIPTOR pSecurityDescriptor,
   LPBOOL lpbDaclPresent,
   PACL *pAcl,
   LPBOOL lpbDaclDefaulted)
```

The parameters are nearly identical to those of SetSecurityDescriptor-Dacl except that the flags are returned to indicate whether a discretionary ACL is actually present and was set as a default or by a user.

To interpret an ACL, first find out how many ACEs it contains.

```
BOOL GetAclInformation (
   PACL pAcl,
   LPVOID pAclInformation,
   DWORD cbAclInfo,
   ACL_INFORMATION_CLASS dwAclInfoClass)
```

In most cases, the ACL information class, dwAclInfoClass, is AclSize-Information, and the pAclInformation parameter is a structure of type ACL_SIZE_INFORMATION. AclRevisionInformation is the other value for the class.

An ACL_SIZE_INFORMATION structure has three members: the most important one is AceCount, which shows how many entries are in the list. To determine whether the ACL is large enough, look at the AclBytesInUse and AclBytesFree members of the ACL_SIZE_INFORMATION structure.

The GetAce function retrieves ACEs by index.

```
BOOL GetAce (
   PACL pAcl,
   DWORD dwAceIndex,
   LPVOID *pAce)
```

Obtain the ACEs (the total number is now known) by using an index. pAce points to an ACE structure, which has a member called Header, which, in turn, has an AceType member. Test the ACE type for ACCESS_ALLOWED_ACE and ACCESS_DENIED_ACE.

Example: Reading File Permissions

Program 15–4 is the function ReadFilePermissions, which Programs 15–1 and 15–2 use. This program methodically uses the preceding functions to extract the information. Its correct operation depends on the fact that the ACL was created by Program 15–3. The function is in the same source module as Program 15–3, so the definitions are not repeated.

Program 15–4 ReadFilePermissions**:** Reading Security Attributes

```
DWORD ReadFilePermissions (LPTSTR lpFileName, LPTSTR userName,
         LPTSTR groupName)
/* Return the UNIX style permissions for a file. */
{
   PSECURITY_DESCRIPTOR pSD = NULL;
   DWORD lenNeeded, permissionBits, iAce;
   BOOL fileDacl, aclDefaulted, ownerDaclDefaulted,
         groupDaclDefaulted;
   DWORD dacl[ACL_SIZE/sizeof(DWORD)];/* ACLS need DWORD alignment */
   PACL pAcl = (PACL) &dacl;
   ACL_SIZE_INFORMATION aclSizeInfo;
   PACCESS_ALLOWED_ACE pAce;
   BYTE aclType;
   PSID pOwnerSid, pGroupSid;
   TCHAR refDomain[2][DOM_SIZE];
   DWORD refDomainCount[2] = {DOM_SIZE, DOM_SIZE};
   DWORD accountNameSize[2] = {ACCT_NAME_SIZE, ACCT_NAME_SIZE};
   SID_NAME_USE sNamUse[] = {SidTypeUser, SidTypeGroup};

   /* Get the required size for the security descriptor. */
   GetFileSecurity (lpFileName, OWNER_SECURITY_INFORMATION |
         GROUP_SECURITY_INFORMATION | DACL_SECURITY_INFORMATION,
         NULL, 0, &lenNeeded);

   /* Create a security descriptor. */
   pSD = malloc (lenNeeded);
   GetFileSecurity (lpFileName, OWNER_SECURITY_INFORMATION |
         GROUP_SECURITY_INFORMATION | DACL_SECURITY_INFORMATION,
         pSD, lenNeeded, &lenNeeded);
   GetSecurityDescriptorDacl (pSD, &fileDacl, &pAcl, &aclDefaulted);
   /* Get the number of ACEs in the ACL. */
   GetAclInformation (pAcl, &aclSizeInfo,
         sizeof (ACL_SIZE_INFORMATION), AclSizeInformation);

   /* Get Each Ace. We know that this ACL was created by
      our InitializeUnixSA function, so the ACES are in the
      same order as the UNIX permission bits. */
   permissionBits = 0;
   for (iAce = 0; iAce < aclSizeInfo.AceCount; iAce++) {
      GetAce (pAcl, iAce, &pAce);
      aclType = pAce->Header.AceType;
      if (aclType == ACCESS_ALLOWED_ACE_TYPE)
         permissionBits |= (0x1 << (8-iAce));
   }

   /* Find the name of the owner and owning group. */
   /* Find the SIDs first. */

   GetSecurityDescriptorOwner (pSD, &pOwnerSid, &ownerDaclDefaulted);
   GetSecurityDescriptorGroup (pSD, &pGroupSid, &groupDaclDefaulted);
   LookupAccountSid (NULL, pOwnerSid, userName, &accountNameSize[0],
         refDomain[0],&refDomainCount[0], &sNamUse[0]);
   LookupAccountSid (NULL, pGroupSid, groupName, &accountNameSize[1],
         refDomain[1],&refDomainCount[1], &sNamUse[1]);
   free (pSD);
   return permissionBits;
}
```

Example: Changing File Permissions

Program 15–5 completes the collection of file security functions. This function, `ChangeFilePermissions`, replaces the existing security descriptor with a new one, preserving the user and group SIDs but creating a new discretionary ACL.

Program 15–5 `ChangeFilePermissions`: Changing Security Attributes

```
BOOL ChangeFilePermissions (DWORD fPerm, LPTSTR fileName,
                   LPDWORD allowedAceMasks, LPDWORD deniedAceMasks)
/* Change permissions in an existing file. The group is unchanged. */
/* Strategy:
   1. Obtain the existing security descriptor using
      the internal function ReadFilePermissions.
   2. Create a security attribute for the owner and permission bits.
   3. Extract the security descriptor.
   4. Set the file security with the new descriptor. */
{
   TCHAR userName[ACCT_NAME_SIZE], groupName[ACCT_NAME_SIZE];
   LPSECURITY_ATTRIBUTES pSA;
   PSECURITY_DESCRIPTOR pSD = NULL;

   ReadFilePermissions (fileName, userName, groupName);
   pSA = InitializeUnixSA (fPerm, userName, groupName,
         allowedAceMasks, deniedAceMasks);
   pSD = pSA->lpSecurityDescriptor;
   SetFileSecurity (fileName, DACL_SECURITY_INFORMATION, pSD);
   return TRUE;
}
```

Securing Kernel and Communication Objects

The preceding sections were concerned mostly with file security, and the same techniques apply to other filelike objects, such as named pipes (Chapter 11), and to kernel objects. Program 15–6, the next example, deals with named pipes, which can be treated in much the same way as files.

Securing Named Pipes

While the code is omitted in the Program 11–3 listing, the server (whose full code appears in the *Examples* file) optionally secures its named pipe to prevent access by unauthorized clients. Optional command line parameters specify the user and group name.

```
Server [UserName GroupName]
```

If the user and group names are omitted, default security is used. Note that the full version of Program 11–3 (in the *Examples* file) and Program 15–6 use techniques from Program 15–3 to create the optional security attributes. However, rather than calling `InitUnixSA`, we now use a simpler function, `Initialize-AccessOnlySA`, which only creates access-allowed ACEs. Program 15–6 shows the relevant code sections that were not shown in Program 11–3. The important security rights for named pipes are follows:

- `FILE_GENERIC_READ`
- `FILE_GENERIC_WRITE`

These two values provide `SYCHRONIZE` rights. The server in Program 15–6 optionally secures its named pipe instances using these rights. Only clients executed by the owner have access, although it would be straightforward to allow group members to access the pipe as well.

Program 15–6 `ServerNP`: Securing a Named Pipe

```
/* Chapter 15. ServerNP. With named pipe security.
 * Multithreaded command line server. Named pipe version.
 * Usage: Server [UserName GroupName]. */
. . .
_tmain (int argc, LPTSTR argv[])
{
   . . .
   HANDLE hNp, hMonitor, hSrvrThread[MAX_CLIENTS];
   DWORD iNp;
   DWORD aceMasks[] =  /* Named pipe access rights */
            {FILE_GENERIC_READ | FILE_GENERIC_WRITE, 0, 0 };

   LPSECURITY_ATTRIBUTES pNPSA = NULL;
   . . .
   if (argc == 4)      /* Optional pipe security. */
      pNPSA = InitializeAccessOnlySA (0440, argv[1], argv[2],
         aceMasks, &hSecHeap);
   . . .
   for (iNp = 0; iNp < MAX_CLIENTS; iNp++) {
      hNp = CreateNamedPipe (SERVER_PIPE, PIPE_ACCESS_DUPLEX,
            PIPE_READMODE_MESSAGE | PIPE_TYPE_MESSAGE | PIPE_WAIT,
            MAX_CLIENTS, 0, 0, INFINITE, pNPSA);

      if (hNp == INVALID_HANDLE_VALUE)
         ReportError (_T ("Failure to open named pipe."), 1, TRUE);
   . . .
}
```