Kernel and Private Object Security

Many objects, such as processes, threads, and mutexes, are *kernel objects*. To get and set kernel security descriptors, use `GetKernelObjectSecurity` and `SetKernelObjectSecurity`, which are similar to the file security functions described in this chapter. However, you need to know the access rights appropriate to an object; the next subsection shows how to find the rights.

It is also possible to associate security descriptors with private, programmer-generated objects, such as a proprietary database. The appropriate functions are `GetPrivateObjectSecurity` and `SetPrivateObjectSecurity`. The programmer must take responsibility for enforcing access and must provide security descriptors with calls to `CreatePrivateObjectSecurity` and `DestroyPrivateObjectSecurity`.

ACE Mask Values

The "user, group, everyone" model that `InitUnixSA` implements will be adequate in many cases, although different models are possible using the same basic techniques.

It is necessary, however, to determine the actual ACE mask values appropriate for a particular kernel object. The values are not always well documented, but there are several ways to determine the values for different kernel objects.

- Read the documentation for the open call for the object in question. The access flags are the same as the flags in the ACE mask. For example, `OpenMutex` uses `MUTEX_ALL_ACCESS` and `SYNCHRONIZE` (the second flag is required for any object that can be used with `WaitForSingleObject` or `WaitForMultipleObjects`). Other objects, such as processes, have many additional access flags.
- The "create" documentation may also supply useful information.
- Inspect the header files `winnt.h` and `winbase.h` for flags that apply to the object.

Example: Securing a Process and Its Threads

The `OpenProcess` documentation shows a fine-grained collection of access rights, which is appropriate considering the various functions that can be performed on a process handle. For example, `PROCESS_TERMINATE` access is required on a process handle in order for a process (actually, a thread within that process) to terminate the process that the handle represents. `PROCESS_QUERY_INFORMATION` access is required in order to perform `GetExitCodeProcess` or `GetPriority-`

`Class` on a process handle. `PROCESS_ALL_ACCESS` permits all access, and `SYNCHRONIZE` access is required to perform a wait function.

To illustrate these concepts, `JobShellSecure.c` upgrades Chapter 6's `JobShell` job management program so that only the owner (or administrator) can access the managed processes. The program is in the *Examples* file.

Overview of Additional Security Features

There is much more to Windows security, but this chapter is an introduction, showing how to secure Windows objects using the security API. The following sections give a brief overview of additional security subjects that some readers will want to explore.

Removing ACEs

The `DeleteAce` function deletes an ACE specified by an index, in a manner similar to that used with `GetAce`.

Absolute and Self-Relative Security Descriptors

Program 15–5, which changed ACLs, had the benefit of simply replacing one security descriptor (SD) with another. To change an existing SD, however, some care is required because of the distinction between absolute (ASD) and self-relative SDs (SRSD). The internal details of these data structures are not important for our purposes, but it is important to understand why there are two distinct SD types and how to convert between them.

- During construction, an SD is absolute, with pointers to various structures in memory. `InitializeSecurityDescriptor` creates an absolute SD. An absolute SD cannot be associated with a permanent object, such as a file, because the structure refers to memory addresses. However, an absolute SD is easy to modify and is fine for a process, thread, event, or other object that is not persistent and is represented by in-memory data structures.
- When the SD is associated with a permanent object, Windows consolidates the SD into a compact "self-relative" structure (SRSD) that can be associated with the object in the file system.
- An SRSD is more compact and more appropriate to pass as a function argument, but it is difficult to change.

- It is possible to convert between the two forms using Windows functions for that purpose. Use `MakeAbsoluteSD` to convert an SRSD, such as the one returned by `GetFileSecurity`. Modify the ASD and then use `MakeSelfRelativeSD` to convert it back. `MakeAbsoluteSD` is one of the more formidable Windows functions, having 11 parameters: two for each of the four SD components, one each for the input and output SDs, and one for the length of the resulting absolute SD.

`InitializeUnixSA` constructs a SA containing multiple ASDs. Exercise 15–16 suggests using only SRSDs.

System ACLs

There is a complete set of functions for managing system ACLs; only system administrators can use it. System ACLs specify which object accesses should be logged. The principal function is `AddAuditAccessAce`, which is similar to `AddAccessAllowedAce`. There is no concept of access denied with system ACLs.

Two other system ACL functions are `GetSecurityDescriptorSacl` and `SetSecurityDescriptorSacl`. These functions are comparable to their discretionary ACL counterparts, `GetSecurityDescriptorDacl` and `SetSecurityDescriptorDacl`.

Access Token Information

Program 15–1 did not solve the problem of obtaining the groups associated with a process in its access token. Program 15–1 simply required the user to specify the group name. You use the `GetTokenInformation` function for this, providing a process handle (Chapter 6). Exercise 15–12 addresses this issue, providing a hint toward the solution. The solution code is also included in the *Examples* file.

SID Management

The examples obtained SIDs from user and group names, but you can also create new SIDs with the `AllocateAndInitializeSid` function. Other functions obtain SID information, and you can even copy (`CopySid`) and compare (`CompareSid`) SIDs.

Summary

Windows implements an extensive security model that goes beyond the one offered by standard UNIX. Programs can secure all objects, not just files. The example programs have shown how to emulate the UNIX permissions and ownership that are set with the `umask`, `chmod`, and `chown` functions. Programs can also set the owner (group and user). The emulation is not easy, but the functionality is much more powerful. The complexity reflects the complexity of the requirements.

Looking Ahead

This chapter completes our presentation of the Windows API.

Additional Reading

Windows

Microsoft Windows Security Resource Kit, Second Edition, by Smith, Komar, and the Microsoft Security Team, and *Microsoft Windows Server 2003 PKI and Certificate Security,* by Brian Komar, provide in depth coverage.

Windows Design and Architecture

Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition, by Solomon, Russinovich, and Ionescu, describes details of Windows security internal implementation.

Common Criteria

See www.commoncriteriaportal.org/thecc.html for information about the Common Criteria levels and the Common Criteria Recognition Agreement.

Exercises

15–1. Extend Program 15–1 so that multiple groups have their own unique permissions. The group name and permission pairs can be separate arguments to the function.

15–2. Extend Program 15–4 so that it can report on all the groups that have ACEs in the object's security descriptor.

15–3. Confirm that `chmodW` has the desired effect of limiting file access.

15–4. Investigate the default security attributes you get with a file.

15–5. What are some of the other access masks you can use with an ACE? The Microsoft documentation supplies some information.

15–6. Enhance both `chmodW` and `lsFP` so that they produce an error message if asked to deal with a file on a non-NTFS file system. `GetVolumeInformation` is required.

15–7. Enhance the `chmodW` command so that there is an `-o` option to set the owning user to be the user of the `chmodW` program.

15–8. Determine the actual size of the ACL buffer that Program 15–3 needs to store the ACEs. Program 15–3 uses 1,024 bytes. Can you determine a formula for estimating the required ACL size?

15–9. The Cygwin Web site (www.cygwin.com) provides an excellent open source Linux-like environment on Windows with a shell and implementations of commands including `chmod` and `ls`. Install this environment and compare the implementations of these two commands with the ones developed here. For example, if you set file permissions using the Cygwin command, does `lsFP` properly show the permissions, and conversely? Compare the Cygwin source code with this chapter's examples to contrast the two approaches to using Windows security.

15–10. The compatibility library contains functions `_open` and `_unmask`, which manage file permissions. Investigate their emulation of UNIX file permissions and compare it with the solutions in this chapter.

15–11. Write a command, `whoami`, that will display your logged-in user name.

15–12. Program 15–3, which created a security descriptor, required the programmer to supply the group name. Modify the program so that it creates permissions for all the user's groups. *Hint:* Use the `OpenProcessToken` function, which returns an array with the group names, although you will need to experiment to find out how the array stores group names. The source program in the *Examples* file contains a partial solution.

15–13. Note in the client/server system that the clients can access exactly the same files and other objects that are available to the server on the server's machine with the server's access rights. Remove this limitation by implementing *security delegation* using the functions `ImpersonateNamedPipeClient` and `RevertToSelf`. Clients that are not in the group used to secure the pipe cannot connect to the server.

15–14. There are several additional Windows functions that you may find useful and that could be applied to simplify or improve this chapter's examples. Look up the following functions: `AreAllAccessesGranted`, `AreAnyAccessesGranted`, `AccessCheck`, and `MapGenericMask`. Can you use these functions to simplify or improve the examples?

15–15. `chmodW` (Program 15–1) calls `GetUserName`, and a code comment suggests an alternative, getting the SID from the current token, to avoid an impersonation problem. Implement and test that change.

15–16. `InitializeUnixSA` uses a heap to simplify destroying the SA structure. An alternative, and arguably superior, method would be to convert all the SDs to be self-relative, use normal (`malloc`) allocations, and use `free`. However, if `InitializeUnixSA` fails before completing, be sure to free the memory that has been allocated.

APPENDIX

A Using the Sample Programs

The book's support Web site (www.jmhartsoftware.com) contains a zip file (the *Examples* file) with the source code for all the sample programs as well as the include files, utility functions, projects, and executables. A number of programs illustrate additional features and solve specific exercises, although the *Examples* file does not include solutions for all exercises or show every alternative implementation.

- All programs have been tested on Windows 7, Vista, XP, Server 2008, and Server 2003 on a wide variety of systems, ranging from laptops to servers. Where appropriate, they have also been tested at one time or another under Windows 9x, although many programs, especially those from later chapters, will not run on Windows 9x or even on NT 4.0, which is also obsolete.
- With a few minor exceptions, nearly all programs compile without warning messages under Microsoft Visual Studio 2005 and 2008 using warning level 3. Visual Studio 2010 (a beta version) easily converted several programs.
- Distinct project directories are provided for Microsoft Visual Studio 2005 and 2008 (32- and 64-bit). The three project directories are `Projects2005`, `Projects2008`, and `Projects2008_64`. The projects build the executable programs in the `run2005`, `run2008`, and `run2008_64` directories, respectively. VS 2010 project and run directories will appear in an updated *Examples* file after VS 2010 is released.
- There is a separate zip file with Visual Studio C++ 6.0 and 7.0 projects; some readers may find these projects convenient, but they are not up to date.
- The generic C library functions are used extensively, as are compiler-specific keywords such as `__try`, `__except`, and `__leave`. The multithreaded C runtime library, `_beginthreadex`, and `_endthreadex` are essential starting with Chapter 7.

- The projects are in release, not debug, form. The projects are all very simple, with minimal dependencies, and can also be created quickly with the desired configuration and as either debug or release versions.
- The projects are defined to build all programs, with the exception of static or dynamic libraries, as *console* applications.

You can also build the programs using open source development tools, such as `gcc` and `g++` in the Gnu Compiler Collection (http://gcc.gnu.org/). Readers interested in these tools should look at the MinGW open source project (www.mingw.org), which describes MinGW as "a port of the GNU Compiler Collection (GCC), and GNU Binutils, for use in the development of native Microsoft Windows applications." I have tested only a few of the programs using these tools, but I have had considerable success using MinGW and have even been able to cross-build, constructing Windows executable programs and DLLs on a Linux system. Furthermore, I've found that `gcc` and `g++` provide very useful 64-bit warning and error messages.

Examples File Organization

The primary directory is named `WSP4_Examples` ("Windows System Programming, Edition 4 Examples"), and this directory can be copied directly to your hard disk. There is a source file subdirectory for each chapter. All include files are in the `Include` directory, and the `Utility` directory contains the common functions such as `ReportError`. Complete projects are in the project directories. Executables and DLLs for all projects are in the run directories.

Download `WindowsSmpEd3` ("Windows Sample Programs, Edition 3") if you want to use Visual Studio 6 or Visual Studio 7.

ReadMe.txt

Everything else you need to know is in the `ReadMe.txt` file, where you will find information about:

- The directories and their contents
- The source code, chapter by chapter
- The include files
- Utility functions

APPENDIX

B Source Code Portability: Windows, UNIX, and Linux

A common, but not universal, application requirement, especially for server applications, is that the application must run on some combination of Windows, Linux, and UNIX.[1] This requirement leads to the need for "source code portability" whereby:

- There is a single set of source code for all target platforms.
- Macros and build parameters define the target platform, which could be any of the three operating systems, along with choices of processor architecture, 32-bit or 64-bit, and operating system vendor.
- Conditional compilation statements, while unavoidable, should be minimal, and the bulk of the source code should be the same for all target platforms.
- The source code is compatible with a wide variety of compilers.
- Performance, resource requirements, and other application characteristics should be similar for all targets.

With sufficient care and some exceptions, you can meet these requirements and build nontrivial source code portable applications. This discussion assumes, however, that there is no GUI interface. This appendix starts by describing some

[1] More precisely, "UNIX" means the POSIX functions specified in *The Single UNIX Specification* (www.opengroup.org/onlinepubs/007908799/). UNIX and Linux implement this specification. In turn, the specification has its historical origins in UNIX.

techniques that have been successful and have met all the requirements. There are, of course, other ways to achieve source code portability beyond what is described here.

The discussion is limited to the POSIX API that is comparable to the topics in this book, and the discussion is not concerned with the complete POSIX environment. However, it's worth pointing out that Cygwin (www.cygwin.com) provides an excellent open source set of POSIX commands and utilities for Windows.

After the discussion are tables that list the Windows functions and their POSIX equivalents, if any. The tables are organized by chapter.

Source Code Portability Strategies

There are several ways to tackle this problem, although there is no single strategy suitable for all functionality areas. Viable strategies, not always mutually exclusive, include:

- Use libraries, possibly open source or Microsoft-provided, that emulate POSIX functions on Windows. This strategy is common and successful, with some examples in this appendix.
- Use libraries, possibly open source, that emulate Windows functions on UNIX/Linux. This strategy is rare, and there are no examples here.
- Use industry-standard functions that Microsoft supports directly. This is also a common and successful strategy for some functionality.
- Use macros instead of libraries to emulate one OS under the other. This strategy is also rare, but there is one example in this appendix.

Windows Services for UNIX

Windows Services for UNIX (SFU) is a Microsoft product that provides a UNIX subsystem, also called Interix, for Windows. The subsystem is implemented in user space on the NT kernel. The current version is 3.5, and you can download it from the Microsoft Web site.

In principle, SFU should satisfy all the requirements. Unfortunately, at publication time, Microsoft plans to discontinue support (the plans were announced in 2005). For example, the Web site states that the supported operating systems are "Windows 2000; Windows 2000 Service Pack 3; Windows 2000 Service Pack 4; Windows Server 2003; Windows XP." Furthermore, "the product will not install on Windows 9x or Windows XP Home Edition or Windows Vista. The product should not be installed on Windows Server 2003 R2. This is an unsupported configuration."

The Wikipedia entry (http://en.wikipedia.org/wiki/Microsoft_Windows_Services_for_UNIX) says, "SFU will be available for download until 2009; general support will continue until 2011; extended support until 2014," and citations to the trade press support this statement.

Source Code Portability for Windows Functionality

The following sections describe some techniques, arranged by chapter order. Some areas are easier than others, and in some cases, there are no straightforward solutions.

File and Directory Management

The Standard C library (CLIB) will support normal file I/O without significant performance impact. However, CLIB does not provide directory management, among other limitations.

Another possible solution is to use the normal POSIX functions, such as `open()` and `read()` and the corresponding Windows functions (see Chapter 1 and the `cpUC` project), such as `_open()` and `_read()`. A simple header file, in the *Examples* file, allows you to use the POSIX function names in the source code. The header file also includes definitions to support time, file attributes, and file locking.

There is no POSIX equivalent to the registry.

Exception and Signal Handling

POSIX signal handling is difficult to emulate in Windows except for a few special cases described in Chapter 4. However, you can write in C++ rather than C and use C++ exception handling rather than Structured Exception Handling (SEH) to achieve some of the requirements.

Memory Management and Memory-Mapped File I/O

There are several aspects to portable memory management code.

- The CLIB functions `malloc`, `calloc`, `realloc`, and `free` are sufficient in most cases for application memory management.
- POSIX does not have functions equivalent to the Windows heap management functions, and Chapter 5 describes some heap management advantages. There are, however, open source solutions, available on Windows and UNIX/Linux, that provide the heap management benefits. One such solution is

"Hoard: A Scalable Memory Allocator for Multithreaded Applications" (www.cs.umass.edu/~emery/hoard/asplos2000.pdf).

- Memory-mapped file I/O provides similar performance and programming simplicity advantages on all operating systems. The Web site's `wcMT` (multithreaded word count) example includes portable file memory-mapping code that has been tested on multiple target systems.

Process Management

Windows `CreateProcess` allows you to emulate the POSIX `fork-exec` sequence, as described in Chapter 6. The principal difficulties arise in:

- Emulating the various `exec` options to specify the command line and environment variables
- Passing handles to the child process
- Managing the parent-child relationships, which Windows does not support

I am not aware of a good open source solution to the process management problem. However, it's worth mentioning that I've successfully developed a library, usable from all OSs, that provides a significant subset of the POSIX process management functionality. This subset was sufficient for the project needs. However, the code was developed under nondisclosure, so it's not in the *Examples* file. Suffice it to say, however, that the task was not difficult and required about two days of work.

Thread Management and Synchronization

Thread management and synchronization portability, at first sight, may seem to be intractable, considering the need for correct operation on a wide variety of platforms. Fortunately, it is not difficult at all. Here is one successful approach.

- Develop your source code using the Pthreads API (www.unix.org/version3/ieee_std.html)
- Use the open source Pthreads library for Windows (http://sources.redhat.com/pthreads-win32/)

This open source library provides good performance, compatible with native Windows code. However, at publication time, it does not yet support slim reader/