Window Stations and Desktops

What do we have in this session?

Window Stations
Desktops
Window Station and Desktop Creation
Process Connection to a Window Station
Thread Connection to a Desktop
Window Station Security and Access Rights
Desktop Security and Access Rights
Window Station and Desktop Reference
CreateProcessAsUser() Window Stations and Desktops Program Example
Windows 2000 and Windows XP
Starting an Interactive Client Process in C++ Working Program Example

Windows provides three main categories of objects:

1. User interface
2. Graphics device interface (GDI), and
3. Kernel

Kernel objects are securable, while user objects and GDI objects are not. Therefore, to provide additional security, user interface objects are managed using window stations and desktops, which themselves are securable objects.
A **window station** is a securable object that is associated with a process, and contains a clipboard, an atom table, and one or more desktop objects. A **desktop** is a securable object contained within a window station. A desktop has a logical display surface and contains user interface objects such as windows, menus, and hooks.

Window Stations

A window station contains a clipboard, an atom table, and one or more desktop objects. Each window station object is a securable object. When a window station is created, it is associated with the calling process and assigned to the current session.
The **interactive window station**, **Winsta0**, is the only window station that can display a user interface or receive user input. It is assigned to the logon session of the **interactive user**, and contains the keyboard, mouse, and display device. **All other window stations are non-interactive**, which means they cannot display a user interface or receive user input.

When a user logs on to a Terminal Services-enabled computer, a session is started for the user. Each session is associated with its own interactive window station.

Desktops

A desktop has a logical display surface and contains user interface objects such as windows, menus, and hooks; it can be used to create and manage windows. Each desktop object is a securable object. When a desktop is created, it is associated with the current window station of the calling process and assigned to the calling thread.
Messages can be sent only between processes that are on the same desktop. In addition, the hook procedure of a process running on a particular desktop can only receive messages intended for windows created in the same desktop.
The desktops associated with the interactive window station, **Winsta0**, can be made to display a user interface and receive user input, but only one of these desktops at a time is active. This active desktop, also known as the **input desktop**, is the one that is currently visible to the user and that receives user input. Applications can use the OpenInputDesktop() function to get a handle to the input desktop. Applications that have the required access can use the SwitchDesktop() function to specify a different input desktop. By default, there are three desktops in the interactive window station:

1. default
2. screen-saver, and
3. Winlogon

The screen-saver and Winlogon desktops are secured desktops. The default desktop is created when the logged-on user starts a process. At that point, the default desktop becomes active, and it is used to interact with the user. Whenever the screen saver activates, the system automatically switches to the screen-saver desktop, which protects the processes on the default desktop from unauthorized users. Unsecured screen savers run on **Winsta0\default**.
The Winlogon desktop is active while a user logs on. The system switches to the default desktop when the shell indicates that it is ready to display something. During the user's session, the system switches to the Winlogon desktop when the user presses the CTRL+ALT+DEL key sequence, or when the User Account Control (UAC) dialog box is open.
For Windows Server 2003 and Windows XP/2000, the UAC dialog box is not supported.
Applications cannot access the Winlogon desktop. Also, applications cannot switch to a different desktop while the Winlogon desktop is active.

Window Station and Desktop Creation

The system automatically creates the interactive window station. When an interactive user logs on, the system associates the interactive window station with the user logon session. The system also

creates the default input desktop for the interactive window station (**Winsta0\default**). Processes started by the logged-on user are associated with the Winsta0\default desktop.
A process can use the CreateWindowStation() function to create a new window station, and the CreateDesktop() or CreateDesktopEx() function to create a new desktop. The number of desktops that can be created is **limited by the size of the system desktop heap**.
When a noninteractive process such as a service application attempts to connect to a window station and no window station exists for the process logon session, the system attempts to create a window station and desktop for the session. The name of the created window station is based on the logon session identifier, and the desktop is named default, as described here:

1. If a service is running in the security context of the LocalSystem account but does not include the SERVICE_INTERACTIVE_PROCESS attribute, it uses the following window station and desktop: **Service-0x0-3e7$\default**. This window station is not interactive, so the service cannot display a user interface. In addition, processes created by the service cannot display a user interface.
2. If the service is running in the security context of a user account, the name of the window station is based on the user SID **Service-0xZ1-Z2$**, where Z1 is the high part of the logon SID and Z2 is the low part of the logon SID. Because a SID is unique to the logon session, two services running in the same security context receive unique window stations. These window stations are not interactive.

The discretionary access control list (DACL) for the window station and desktop includes the following access rights for the service's user account:

Window Station:

1. WINSTA_ACCESSCLIPBOARD
2. WINSTA_ACCESSGLOBALATOMS
3. WINSTA_CREATEDESKTOP
4. WINSTA_EXITWINDOWS
5. WINSTA_READATTRIBUTES
6. STANDARD_RIGHTS_REQUIRED

Desktop:

1. DESKTOP_CREATEMENU
2. DESKTOP_CREATEWINDOW
3. DESKTOP_ENUMERATE
4. DESKTOP_HOOKCONTROL
5. DESKTOP_READOBJECTS
6. DESKTOP_WRITEOBJECTS

7. STANDARD_RIGHTS_REQUIRED

Process Connection to a Window Station

A process automatically establishes a connection to a window station and desktop when it first calls a USER32 or GDI32 function (other than the window station and desktop functions). The system determines the window station to which a process connects according to the following rules:

1. If the process has called the SetProcessWindowStation() function, it connects to the window station specified in that call.
2. If the process did not call SetProcessWindowStation(), it connects to the window station inherited from the parent process.
3. If the process did not call SetProcessWindowStation() and did not inherit a window station, the system attempts to open for MAXIMUM_ALLOWED access and connect to a window station as follows:

    a. If a window station name was specified in the lpDesktop member of the STARTUPINFO structure that was used when the process was created, the process connects to the specified window station.
    b. Otherwise, if the process is running in the logon session of the interactive user, the process connects to the interactive window station.
    c. If the process is running in a noninteractive logon session, the window station name is formed based on the logon session identifier and an attempt is made to open that window station. If the open operation fails because this window station does not exist, the system tries to create the window station and a default desktop.

The window station assigned during this connection process cannot be closed by calling the CloseWindowStation() function.
When a process is connecting to a window station, the system searches the process's handle table for inherited handles. The system uses the first window station handle it finds. If you want a child process to connect to a particular inherited window station, you must ensure that the only the desired handle is marked inheritable. If a child process inherits multiple window station handles, the results of the window station connection are undefined.
Handles to a window station that the system opens while connecting a process to a window station are not inheritable.

Thread Connection to a Desktop

After a process connects to a window station, the system assigns a desktop to the thread making the connection. The system determines the desktop to assign to the thread according to the following rules:

1. If the thread has called the SetThreadDesktop() function, it connects to the specified desktop.
2. If the thread did not call SetThreadDesktop(), it connects to the desktop inherited from the parent process.
3. If the thread did not call SetThreadDesktop() and did not inherit a desktop, the system attempts to open for MAXIMUM_ALLOWED access and connect to a desktop as follows:

    a. If a desktop name was specified in the lpDesktop member of the STARTUPINFO structure that was used when the process was created, the thread connects to the specified desktop.
    b. Otherwise, the thread connects to the default desktop of the window station to which the process connected.

The desktop assigned during this connection process cannot be closed by calling the CloseDesktop() function.
When a process is connecting to a desktop, the system searches the process's handle table for inherited handles. The system uses the first desktop handle it finds. If you want a child process to connect to a particular inherited desktop, you must ensure that the only the desired handle is marked inheritable. If a child process inherits multiple desktop handles, the results of the desktop connection are undefined. Handles to a desktop that the system opens while connecting a process to a desktop are not inheritable.

Window Station Security and Access Rights

Security enables you to control access to window station objects. You can specify a security descriptor for a window station object when you call the CreateWindowStation() function. If you specify NULL, the window station gets a default security descriptor. The ACLs in the default security descriptor for a window station come from the primary or impersonation token of the creator.
To get or set the security descriptor of a window station object, call the GetSecurityInfo() and SetSecurityInfo() functions.
When you call the OpenWindowStation() function, the system checks the requested access rights against the object's security descriptor.
The valid access rights for window station objects include the standard access rights and some object-specific access rights. The following table lists the standard access rights used by all objects.

Value	Meaning
DELETE (0x00010000L)	Required to delete the object.
READ_CONTROL (0x00020000L)	Required to read information in the security descriptor for the object, not including the information in the SACL. To read or write the SACL, you must request the ACCESS_SYSTEM_SECURITY access right.

SYNCHRONIZE (0x00100000L)	Not supported for window station objects.
WRITE_DAC (0x00040000L)	Required to modify the DACL in the security descriptor for the object.
WRITE_OWNER (0x00080000L)	Required to change the owner in the security descriptor for the object.

The following table lists the object-specific access rights.

Access right	Description
WINSTA_ALL_ACCESS (0x37F)	All possible access rights for the window station.
WINSTA_ACCESSCLIPBOARD (0x0004L)	Required to use the clipboard.
WINSTA_ACCESSGLOBALATOMS (0x0020L)	Required to manipulate global atoms.
WINSTA_CREATEDESKTOP (0x0008L)	Required to create new desktop objects on the window station.
WINSTA_ENUMDESKTOPS (0x0001L)	Required to enumerate existing desktop objects.
WINSTA_ENUMERATE (0x0100L)	Required for the window station to be enumerated.
WINSTA_EXITWINDOWS (0x0040L)	Required to successfully call the ExitWindows() or ExitWindowsEx() function. Window stations can be shared by users and this access type can prevent other users of a window station from logging off the window station owner.
WINSTA_READATTRIBUTES (0x0002L)	Required to read the attributes of a window station object. This attribute includes color settings and other global window station properties.
WINSTA_READSCREEN (0x0200L)	Required to access screen contents.
WINSTA_WRITEATTRIBUTES (0x0010L)	Required to modify the attributes of a window station object. The attributes include color settings and other global window station properties.

The following are the generic access rights for the interactive window station object, which is the window station assigned to the logon session of the interactive user.

Access right	Description
GENERIC_READ	STANDARD_RIGHTS_READ WINSTA_ENUMDESKTOPS WINSTA_ENUMERATE WINSTA_READATTRIBUTES

	WINSTA_READSCREEN
GENERIC_WRITE	STANDARD_RIGHTS_WRITE WINSTA_ACCESSCLIPBOARD WINSTA_CREATEDESKTOP WINSTA_WRITEATTRIBUTES
GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE WINSTA_ACCESSGLOBALATOMS WINSTA_EXITWINDOWS
GENERIC_ALL	STANDARD_RIGHTS_REQUIRED WINSTA_ACCESSCLIPBOARD WINSTA_ACCESSGLOBALATOMS WINSTA_CREATEDESKTOP WINSTA_ENUMDESKTOPS WINSTA_ENUMERATE WINSTA_EXITWINDOWS WINSTA_READATTRIBUTES WINSTA_READSCREEN WINSTA_WRITEATTRIBUTES

The following are the generic access rights for a noninteractive window station object. The system assigns noninteractive window stations to all logon sessions other than that of the interactive user.

Access right	Description
GENERIC_READ	STANDARD_RIGHTS_READ WINSTA_ENUMDESKTOPS WINSTA_ENUMERATE WINSTA_READATTRIBUTES
GENERIC_WRITE	STANDARD_RIGHTS_WRITE WINSTA_ACCESSCLIPBOARD WINSTA_CREATEDESKTOP
GENERIC_EXECUTE	STANDARD_RIGHTS_EXECUTE WINSTA_ACCESSGLOBALATOMS WINSTA_EXITWINDOWS
GENERIC_ALL	STANDARD_RIGHTS_REQUIRED WINSTA_ACCESSCLIPBOARD WINSTA_ACCESSGLOBALATOMS WINSTA_CREATEDESKTOP WINSTA_ENUMDESKTOPS WINSTA_ENUMERATE WINSTA_EXITWINDOWS WINSTA_READATTRIBUTES

You can request the ACCESS_SYSTEM_SECURITY access right to a window station object if you want to read or write the object's SACL.

Desktop Security and Access Rights

Security enables you to control access to desktop objects. You can specify a security descriptor for a desktop object when you call the CreateDesktop() or CreateDesktopEx() function. If you specify NULL, the desktop gets a default security descriptor. The ACLs in the default security descriptor for a desktop come from its parent window station.
To get or set the security descriptor of a window station object, call the GetSecurityInfo() and SetSecurityInfo() functions. When you call the OpenDesktop() or OpenInputDesktop() function, the system checks the requested access rights against the object's security descriptor.
The valid access rights for desktop objects include the standard access rights and some object-specific access rights. The following table lists the standard access rights used by all objects.

Value	Meaning
DELETE (0x00010000L)	Required to delete the object.
READ_CONTROL (0x00020000L)	Required to read information in the security descriptor for the object, not including the information in the SACL. To read or write the SACL, you must request the ACCESS_SYSTEM_SECURITY access right.
SYNCHRONIZE (0x00100000L)	Not supported for desktop objects.
WRITE_DAC (0x00040000L)	Required to modify the DACL in the security descriptor for the object.
WRITE_OWNER (0x00080000L)	Required to change the owner in the security descriptor for the object.

The following table lists the object-specific access rights.

Access right	Description
DESKTOP_CREATEMENU (0x0004L)	Required to create a menu on the desktop.
DESKTOP_CREATEWINDOW (0x0002L)	Required to create a window on the desktop.
DESKTOP_ENUMERATE (0x0040L)	Required for the desktop to be enumerated.
DESKTOP_HOOKCONTROL (0x0008L)	Required to establish any of the window hooks.
DESKTOP_JOURNALPLAYBACK (0x0020L)	Required to perform journal playback on a desktop.
DESKTOP_JOURNALRECORD (0x0010L)	Required to perform journal recording on a desktop.
DESKTOP_READOBJECTS (0x0001L)	Required to read objects on the desktop.

DESKTOP_SWITCHDESKTOP (0x0100L)	Required to activate the desktop using the SwitchDesktop() function.
DESKTOP_WRITEOBJECTS (0x0080L)	Required to write objects on the desktop.

The following are the generic access rights for a desktop object contained in the interactive window station of the user's logon session.

Access right	Description
GENERIC_READ	DESKTOP_ENUMERATE DESKTOP_READOBJECTS STANDARD_RIGHTS_READ
GENERIC_WRITE	DESKTOP_CREATEMENU DESKTOP_CREATEWINDOW DESKTOP_HOOKCONTROL DESKTOP_JOURNALPLAYBACK DESKTOP_JOURNALRECORD DESKTOP_WRITEOBJECTS STANDARD_RIGHTS_WRITE
GENERIC_EXECUTE	DESKTOP_SWITCHDESKTOP STANDARD_RIGHTS_EXECUTE
GENERIC_ALL	DESKTOP_CREATEMENU DESKTOP_CREATEWINDOW DESKTOP_ENUMERATE DESKTOP_HOOKCONTROL DESKTOP_JOURNALPLAYBACK DESKTOP_JOURNALRECORD DESKTOP_READOBJECTS DESKTOP_SWITCHDESKTOP DESKTOP_WRITEOBJECTS STANDARD_RIGHTS_REQUIRED

You can request the ACCESS_SYSTEM_SECURITY access right to a desktop object if you want to read or write the object's SACL.

Window Station and Desktop Reference

The following elements are used with window stations and desktops:

1. Window Station and Desktop Functions
2. Window Station and Desktop Structures

CreateProcessAsUser() Window Stations and Desktops Program Example

When a process is started by means of the CreateProcessAsUser() function, the process will be started into a window station and desktop combination based on the value of lpDesktop in the STARTUPINFO structure parameter:

1. If a window station and desktop combination is specified in the lpDesktop member, the system will try to start the process into that window station and desktop.
2. If the lpDesktop member is initialized to NULL, the system will try to use the same window station and desktop as the calling process if the system is associated with the interactive window station.
3. If the lpDesktop member is not initialized to NULL, the system will create a new window station and desktop that you cannot see.
4. If the system is initialized with the empty string, "", it will either create a new window station and desktop that you cannot see, or if one has been created by means of a prior call by using the same access token, the existing window station and desktop will be used.

Sometimes the process may fail to start, and one of the following error messages may appear:

Error message 1:

Initialization of the dynamic library <system>\system32\user32.dll failed. The process is terminating abnormally.

Error message 2:

Initialization of the dynamic library <system>\system32\kernel32.dll failed. The process is terminating abnormally.

The error message occurs when the process that is started causes the initialization code in either the User32.dll or the Kernel32.dll file to fail because of an API call from the started process that does not have correct security access to either the targeted window station or desktop. For example, if the process that was started was trying to create a window, the process would have to have DESKTOP_CREATEWINDOW access to the desktop object. If the process has not been granted this access right, an error would occur in the User32.dll file, which would cause the system error box to appear and the process would fail to start.
Note that sometimes the process may start, but fail to draw its GUI correctly. The best method to resolve these and other potential access related problems is to grant the user full access to both the targeted window station and desktop. For example, if you want the process that is started by the CreateProcessAsUser() function to be interactive, specify the following window station and desktop combination:

```
winsta0\default
```

Windows 2000 and Windows XP Issues

A new API was introduced beginning with Windows 2000, CreateProcessWithLogonW(). If the lpDesktop member of the STARTUPINFO structure is initialized to either NULL or "", CreateProcessWithLogonW() implementation adds permissions for the specified user account to the inherited window station and desktop. If the application specifies a desktop in the lpDesktop member, it is the responsibility of the application to add permission for the specified user account to the specified window station and desktop.
The following sample code grants the user named Johnny access to the interactive window station and desktop, "winsta0\\default". Access is granted based on the logon security ID (SID) of the user Johnny. The access control entry (ACE) for each object is based on Johnny's logon SID. The code executes the cmd.exe file.
An application that runs many processes such as a scheduler service may want to remove the new ACE after the process has completed because the ACEs accumulate on the DACL of both the window station and desktop object.
All Microsoft Windows NT, Windows 2000, Windows XP Executive objects, which Window stations and Desktops belong to, have a 2K limit on Access Control Lists (ACL). SetUserObjectSecurity() returns ERROR_NOT_ENOUGH_QUOTA when this limit is reached. This 2K limit equals approximately 84 or 85 Access Control Entries (ACE). SetUserObjectSecurity() returns will return ERROR_NOT_ENOUGH_QUOTA.
It is recommended that you add an ACE based on the Logon Security Identifier (SID) since this duplicates the process used by the system. Consider the following options when you experience this problem:

1. If you are launching many processes running in the same security context or logon session, you might want to add one ACE versus an ACE for every process.
2. If you can keep track of when the process dies, you should remove the ACE when the process has terminated.
3. If you cannot track when the process dies, there are several procedures that you can use to remove any unnecessary ACEs. You can enumerate processes, read the Logon Security Identifier (SID) or User SID from the process token, and compare one of them to the ACEs stored in the DACL for the window station and desktop objects. This depends on which ACE you used to secure the object. Remove any ACEs for processes that are no longer running on the system. NOTE: there might be other processes that are adding ACEs to the objects.
4. If you are launching many processes, you might want to add an ACE based on the processes logon type. For example, this could be either the Interactive or Batch SID. You would not have to add any additional ACEs for processes with the same logon type.

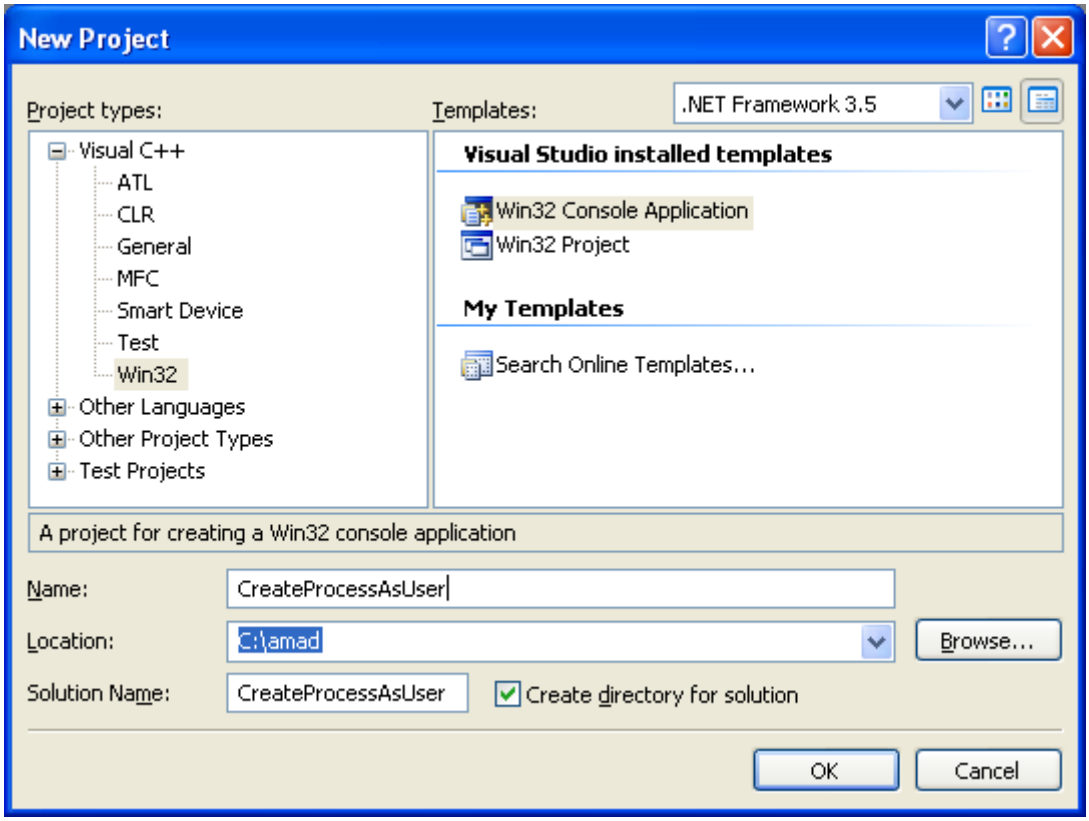
The lpApplicationName parameter can be NULL, in which case the executable name must be the first white space–delimited string in lpCommandLine. If the executable or path name has a space in it, there is a risk that a different executable could be run because of the way the function parses spaces. The following example is dangerous because the function will attempt to run "Program.exe", if it exists, instead of "MyApp.exe".

```
LPTSTR szCmdline[] = _tcsdup(TEXT("C:\\Program Files\\MyApp"));
CreateProcessAsUser(hToken, NULL, szCmdline, /*...*/ );
```

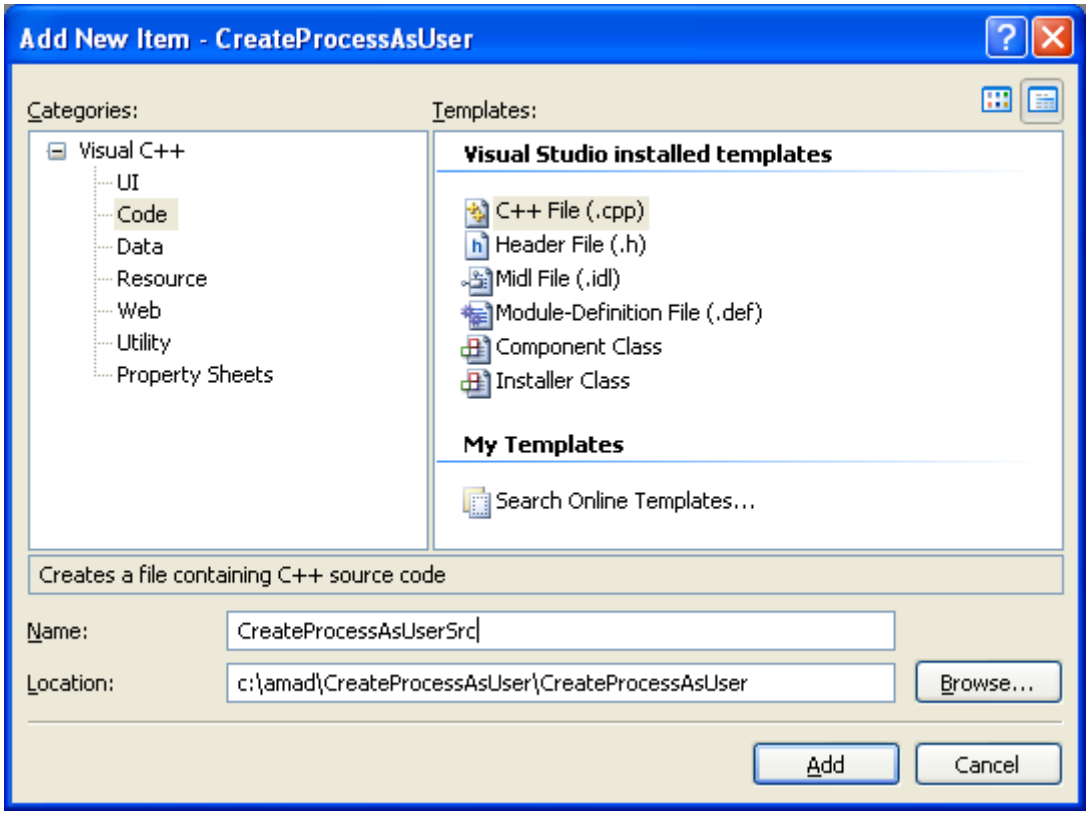
If a malicious user were to create an application called "Program.exe" on a system, any program that incorrectly calls CreateProcessAsUser() using the Program Files directory will run this application instead of the intended application. To avoid this problem, do not pass NULL for lpApplicationName. If you do pass NULL for lpApplicationName, use quotation marks around the executable path in lpCommandLine, as shown in the code snippet example below.

```
LPTSTR szCmdline[] = _tcsdup(TEXT("\"C:\\Program Files\\MyApp\""));
CreateProcessAsUser(hToken, NULL, szCmdline, /*...*/);
```

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.



Then, add the source file and give it a suitable name.



Next, add the following source code.

```
#include <windows.h>
#include <stdio.h>

// Constants
#define WINSTA_ALL (WINSTA_ACCESSCLIPBOARD   | WINSTA_ACCESSGLOBALATOMS | \
                                WINSTA_CREATEDESKTOP | WINSTA_ENUMDESKTOPS |
WINSTA_ENUMERATE | \
                                WINSTA_EXITWINDOWS |WINSTA_READATTRIBUTES    |
WINSTA_READSCREEN |\
                                WINSTA_WRITEATTRIBUTES   | DELETE | READ_CONTROL |
WRITE_DAC | WRITE_OWNER)
#define DESKTOP_ALL (DESKTOP_CREATEMENU | DESKTOP_CREATEWINDOW   | \
                                DESKTOP_ENUMERATE | DESKTOP_HOOKCONTROL | \
                                DESKTOP_JOURNALPLAYBACK | DESKTOP_JOURNALRECORD |
\
                                DESKTOP_READOBJECTS      | DESKTOP_SWITCHDESKTOP |
\
                                DESKTOP_WRITEOBJECTS     | DELETE   |    READ_CONTROL
| \
                                WRITE_DAC   |    WRITE_OWNER)
#define GENERIC_ACCESS (GENERIC_READ | GENERIC_WRITE | GENERIC_EXECUTE |
GENERIC_ALL)

// Prototypes
BOOL ObtainSid(
        HANDLE hToken,  // Handle to an process access token.
```

```
        PSID   *psid    // ptr to the buffer of the logon sid
        );
void RemoveSid(
        PSID *psid      // ptr to the buffer of the logon sid
        );
BOOL AddTheAceWindowStation(
        HWINSTA hwinsta,      // handle to a windowstation
        PSID    psid    // logon sid of the process
        );
BOOL AddTheAceDesktop(
        HDESK hdesk,    // handle to a desktop
        PSID  psid            // logon sid of the process
        );

int wmain(int argc, WCHAR **argv)
{
      HANDLE            hToken;
      HDESK       hdesk;
      HWINSTA           hwinsta, hwinstaold;
      PROCESS_INFORMATION pi;
      PSID  psid;
      STARTUPINFO si;
      DWORD size = 0;
      WCHAR szCommandLine[] = L"C:\\Windows\\system32\\calc.exe";
      // The following failed lol - ...0xC0000005: Access violation writing
location...
      //          LPTSTR szCommandLine = L"C:\\Windows\\system32\\calc.exe";

      // attempts to log a user on to the local computer.
      // obtain an access token for the user Johhny with 123 as the password
      // on the local account database (the dot)
      if (!LogonUser(L"Johnny", L".",L"123",
LOGON32_LOGON_INTERACTIVE,LOGON32_PROVIDER_DEFAULT,&hToken))
      {
            wprintf(L"LogonUser() failed, error %d!\n", GetLastError());
            return 1;
      }
      else
            wprintf(L"LogonUser() is OK!\n");

    // Opens the specified window station.
      // obtain a handle to the interactive windowstation
      hwinsta = OpenWindowStation(
            L"winsta0", // must belong to the current session.
            FALSE,
            READ_CONTROL | WRITE_DAC);

      if (hwinsta == NULL)
      {
            wprintf(L"OpenWindowStation() failed, error %d!\n", GetLastError());
            return 1;
      }
      else
            wprintf(L"OpenWindowStation() is OK!\n");

      // Retrieves a handle to the current window station for the calling
process.
      hwinstaold = GetProcessWindowStation();
```

```
	// Assigns the specified window station to the calling process.
	// set the windowstation to winsta0 so that you obtain the correct default
desktop
	if (!SetProcessWindowStation(hwinsta))
	{
		wprintf(L"SetProcessWindowStation() failed, error %d!\n",
GetLastError());
		return 1;
	}
	else
		wprintf(L"SetProcessWindowStation() is OK!\n");

	// Opens the specified desktop object.
	// obtain a handle to the "default" desktop
	hdesk = OpenDesktop(
		L"default", // desktop must belong to the current window station.
		0,
		FALSE,
		READ_CONTROL | WRITE_DAC | DESKTOP_WRITEOBJECTS |
DESKTOP_READOBJECTS);

	if (hdesk == NULL)
	{
		wprintf(L"OpenDesktop() failed, error %d!\n", GetLastError());
		return 1;
	}
	else
		wprintf(L"OpenDesktop() is pretty fine!\n");

	// obtain the logon sid of the user fester
	if (!ObtainSid(hToken, &psid))
	{
		wprintf(L"ObtainSid() failed, error %d!\n", GetLastError());
		return 1;
	}
	else
		wprintf(L"ObtainSid() is OK!\n");

	// add the user to interactive windowstation
	if (!AddTheAceWindowStation(hwinsta, psid))
	{
		wprintf(L"AddTheAceWindowStation() failed, error %d!\n",
GetLastError());
		return 1;
	}
	else
		wprintf(L"AddTheAceWindowStation() is OK!\n");

	// add user to "default" desktop
	if (!AddTheAceDesktop(hdesk, psid))
	{
		wprintf(L"AddTheAceDesktop() failed, error %d!\n", GetLastError());
		return 1;
	}
	else
		wprintf(L"AddTheAceDesktop() is OK!\n");
	// free the buffer for the logon sid
```

```
	wprintf(L"Removing the SID...\n");
	RemoveSid(&psid);

	// close the handles to the interactive windowstation and desktop
	if(CloseWindowStation(hwinsta) != 0)
		wprintf(L"hwinsta handle was closed successfully!\n");
	else
		wprintf(L"Failed to close hwinsta handle, error %d\n",
GetLastError());
	// The CloseDesktop function will fail if any thread in the calling
process
	// is using the specified desktop handle or if the handle refers to
	// the initial desktop of the calling process
	if(CloseDesktop(hdesk) != 0)
		wprintf(L"hdesk handle was closed successfully!\n");
	else
		wprintf(L"Failed to close hdesk handle, error %d\n",
GetLastError());

	// initialize STARTUPINFO structure
	SecureZeroMemory(&si, sizeof(STARTUPINFO));
	si.cb		= sizeof(STARTUPINFO);
	// the process to be interactive
	si.lpDesktop = L"winsta0\\default";

	SecureZeroMemory(&pi, sizeof(PROCESS_INFORMATION));

	// Creates a new process and its primary thread.
	// The new process runs in the security context of the user represented by
the specified token.
	if (!CreateProcessAsUser(
		hToken,	// handle to the primary token that represents a user
		NULL, // name of the module to be executed.
		szCommandLine,
		// Failed for the following - ...0xC0000005: Access violation
writing location XXXXXX
		//		L"C:\\Windows\\System32\\cmd.exe",  // The command line
to be executed
		NULL, // A pointer to a SECURITY_ATTRIBUTES structure for the new
process object
		NULL, // A pointer to a SECURITY_ATTRIBUTES structure for the new
thread object
		FALSE,	// FALSE, the handles are not inherited.
		NORMAL_PRIORITY_CLASS | CREATE_NEW_CONSOLE,	// flags that
control the priority

	//class and the creation of the process
		NULL, // A pointer to an environment block for the new process.
		NULL, // The full path to the current directory for the process
		&si,  // Pointer to STARTUPINFO structure
		&pi)) // Pointer to PROCESS_INFORMATION structure
	{
		wprintf(L"CreateProcessAsUser() failed, error %d!\n",
GetLastError());
		return 1;
	}
	else
		wprintf(L"CreateProcessAsUser() is pretty OK!\n");
```

```
	// Assigns the specified window station to the calling process
	SetProcessWindowStation(hwinstaold); //set it back

	// close the handles
	if(CloseHandle(pi.hProcess) != 0)
		wprintf(L"pi.hProcess handle was closed successfully!\n");
	else
		wprintf(L"Failed to close pi.hProcess handle! error %d\n",
GetLastError());

	if(CloseHandle(pi.hThread) != 0)
		wprintf(L"pi.hThread handle was closed successfully!\n");
	else
		wprintf(L"Failed to close pi.hThread handle! error %d\n",
GetLastError());
	return 0;
}

BOOL ObtainSid(HANDLE hToken, PSID *psid)
{
	BOOL  bSuccess = FALSE; // assume function will fail
	DWORD dwIndex;
	DWORD dwLength = 0;
	TOKEN_INFORMATION_CLASS tic      = TokenGroups;
	PTOKEN_GROUPS     ptg      = NULL;

	__try
	{
		// determine the size of the buffer
		if (!GetTokenInformation(hToken,tic,(LPVOID)ptg,0,&dwLength))
		{
			if (GetLastError() == ERROR_INSUFFICIENT_BUFFER)
			{
				ptg =
(PTOKEN_GROUPS)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwLength);
				if (ptg == NULL)
				{
					wprintf(L" Heap allocation for ptg is failed,
error %d\n", GetLastError());
					__leave;
				}
				else
					wprintf(L" Heap allocation for ptg is OK!\n");
			}
			else
				__leave;
		}
		else
			wprintf(L" GetTokenInformation() is OK!\n");

		// obtain the groups the access token belongs to
		if (!GetTokenInformation(hToken,tic,(LPVOID)ptg,dwLength,&dwLength))
			__leave;
		else
			wprintf(L" GetTokenInformation() is pretty working!\n");

		 // determine which group is the logon sid
```

```
            for (dwIndex = 0; dwIndex < ptg->GroupCount; dwIndex++)
                  {
                        if ((ptg->Groups[dwIndex].Attributes &
SE_GROUP_LOGON_ID) ==  SE_GROUP_LOGON_ID)
                        {
                              // determine the length of the sid
                              dwLength = GetLengthSid(ptg-
>Groups[dwIndex].Sid);
                              // allocate a buffer for the logon sid
                              *psid =
(PSID)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwLength);
                              if (*psid == NULL)
                              {
                                    wprintf(L" Heap allocation for psid is
failed, error %d\n", GetLastError());
                                    __leave;
                              }
                              else
                                    wprintf(L" Heap allocation for psid is
OK!\n");

                              // obtain a copy of the logon sid
                              if (!CopySid(dwLength, *psid, ptg-
>Groups[dwIndex].Sid))
                              {
                                    wprintf(L" CopySid() failed, error %d\n",
GetLastError());
                                    __leave;
                              }
                              else
                                    wprintf(L" CopySid() is OK!\n");

                              // break out of the loop because the logon sid
has been found
                              break;
                        }
                  }
                  // indicate success
                  bSuccess = TRUE;
      }
      __finally
      {
            // free the buffer for the token group
            if (ptg != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)ptg);
      }
      return bSuccess;
}

void RemoveSid(PSID *psid)
{
      HeapFree(GetProcessHeap(), 0, (LPVOID)*psid);
}

BOOL AddTheAceWindowStation(HWINSTA hwinsta, PSID psid)
{
      ACCESS_ALLOWED_ACE   *pace;
      ACL_SIZE_INFORMATION aclSizeInfo;
```

```
	BOOL  bDaclExist;
	BOOL  bDaclPresent;
	BOOL  bSuccess  = FALSE; // assume function will fail
	DWORD dwNewAclSize;
	DWORD dwSidSize = 0;
	DWORD dwSdSizeNeeded;
	PACL  pacl;
	PACL  pNewAcl;
	PSECURITY_DESCRIPTOR psd      = NULL;
	PSECURITY_DESCRIPTOR psdNew   = NULL;
	PVOID pTempAce;
	SECURITY_INFORMATION si = DACL_SECURITY_INFORMATION;
	unsigned int      i;

	__try
	{
		// obtain the dacl for the windowstation
		// retrieves security information for the specified user object.
		if
(!GetUserObjectSecurity(hwinsta,&si,psd,dwSidSize,&dwSdSizeNeeded))
			if (GetLastError() == ERROR_INSUFFICIENT_BUFFER)
			{
				psd =
(PSECURITY_DESCRIPTOR)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwSdSizeNeeded
);
				if (psd == NULL)
				{
					wprintf(L"  HeapAlloc() for psd failed, error
%d\n", GetLastError());
					__leave;
				}
				else
					wprintf(L"  Heap allocation for psd is OK!\n");

				psdNew =
(PSECURITY_DESCRIPTOR)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwSdSizeNeeded
);
				if (psdNew == NULL)
				{
					wprintf(L"  HeapAlloc() for psdNew failed, error
%d\n", GetLastError());
					__leave;
				}
				else
					wprintf(L"  Heap allocation for psdNew is OK!\n");

				dwSidSize = dwSdSizeNeeded;
				if
(!GetUserObjectSecurity(hwinsta,&si,psd,dwSidSize,&dwSdSizeNeeded))
				{
					wprintf(L"  GetUserObjectSecurity() failed, error
%d\n", GetLastError());
					__leave;
				}
				else
					wprintf(L"  GetUserObjectSecurity() should be
fine!\n");
			}
```

```
                else
                        __leave;

            // create a new dacl
            if
(!InitializeSecurityDescriptor(psdNew,SECURITY_DESCRIPTOR_REVISION))
            {
                wprintf(L"  InitializeSecurityDescriptor() failed, error
%d\n", GetLastError());
                __leave;
            }
            else
                wprintf(L"  InitializeSecurityDescriptor() is nothing
wrong!\n");

            // get dacl from the security descriptor
            if (!GetSecurityDescriptorDacl(psd,&bDaclPresent,&pacl,&bDaclExist))
            {
                wprintf(L"  GetSecurityDescriptorDacl() failed, error %d\n",
GetLastError());
                __leave;
            }
            else
                wprintf(L"  GetSecurityDescriptorDacl() is working!\n");

            // initialize
            ZeroMemory(&aclSizeInfo, sizeof(ACL_SIZE_INFORMATION));
            aclSizeInfo.AclBytesInUse = sizeof(ACL);

            // call only if the dacl is not NULL
            if (pacl != NULL)
            {
                // get the file ACL size info
                if (!GetAclInformation(
                        pacl,
                        (LPVOID)&aclSizeInfo,
                        sizeof(ACL_SIZE_INFORMATION),
                        AclSizeInformation))
                {
                        wprintf(L"  GetAclInformation() failed, error %d\n",
GetLastError());
                        __leave;
                }
                else
                        wprintf(L"  Woww... GetAclInformation() is working!\n");
            }

            // compute the size of the new acl
            dwNewAclSize = aclSizeInfo.AclBytesInUse + (2 *
sizeof(ACCESS_ALLOWED_ACE)) +
                (2 * GetLengthSid(psid)) - (2 * sizeof(DWORD));

             // allocate memory for the new acl
             pNewAcl =
(PACL)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwNewAclSize);
             if (pNewAcl == NULL)
                  {
```

```
                        wprintf(L"  Heap allocation for pNewAcl failed, error
%d\n", GetLastError());
                  __leave;
                   }
                   else
                        wprintf(L"  Heap allocation for pNewAcl is OK!\n");

            // initialize the new dacl
            if (!InitializeAcl(pNewAcl, dwNewAclSize, ACL_REVISION))
                  {
                        wprintf(L"  InitializeAcl() failed, error %d\n",
GetLastError());
                  __leave;
                   }
                   else
                        wprintf(L"  InitializeAcl() is pretty damn OK!\n");

            // if DACL is present, copy it to a new DACL
            if (bDaclPresent) // only copy if DACL was present
                  {
                        // copy the ACEs to our new ACL
                        if (aclSizeInfo.AceCount)
                        {
                              for (i=0; i < aclSizeInfo.AceCount; i++)
                              {
                                    // get an ACE
                                    if (!GetAce(pacl, i, &pTempAce))
                                    {
                                          wprintf(L"  GetAce() failed, error
%d\n", GetLastError());
                                          __leave;
                                    }
                                    else
                                          wprintf(L"  GetAce() is OK!\n");

                                    // add the ACE to the new ACL
                                    if
(!AddAce(pNewAcl,ACL_REVISION,MAXDWORD,pTempAce,((PACE_HEADER)pTempAce)-
>AceSize))
                                    {
                                          wprintf(L"  AddAce() failed, error
%d\n", GetLastError());
                                          __leave;
                                    }
                                    else
                                          wprintf(L"  AddAce() is OK!\n");
                              }
                        }
                  }

                  // add the first ACE to the windowstation
                  pace = (ACCESS_ALLOWED_ACE
*)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,
                        sizeof(ACCESS_ALLOWED_ACE) + GetLengthSid(psid) -
sizeof(DWORD));

                  if (pace == NULL)
                  {
```

```
                        wprintf(L"  Heap allocation for pace failed, error
%d\n", GetLastError());
                        __leave;
                  }
                  else
                        wprintf(L"  Heap allocation for pace is OK!\n");

                  pace->Header.AceType  = ACCESS_ALLOWED_ACE_TYPE;
                  pace->Header.AceFlags = CONTAINER_INHERIT_ACE
|INHERIT_ONLY_ACE |OBJECT_INHERIT_ACE;
                  pace->Header.AceSize  = (WORD)(sizeof(ACCESS_ALLOWED_ACE) +
GetLengthSid(psid) - sizeof(DWORD));
                  pace->Mask            = GENERIC_ACCESS;

                  if (!CopySid(GetLengthSid(psid), &pace->SidStart, psid))
                  {
                        wprintf(L"  CopySid() failed, error %d\n",
GetLastError());
                        __leave;
                  }
                  else
                        wprintf(L"  CopySid() is pretty fine!\n");

                  if (!AddAce(pNewAcl,ACL_REVISION,MAXDWORD,(LPVOID)pace,pace-
>Header.AceSize))
                  {
                        wprintf(L"  AddAce() failed, error %d\n",
GetLastError());
                        __leave;
                  }
                  else
                        wprintf(L"  AddAce() 1 is pretty fine!\n");

            // add the second ACE to the windowstation
                  pace->Header.AceFlags = NO_PROPAGATE_INHERIT_ACE;
                  pace->Mask            = WINSTA_ALL;

                  if (!AddAce(pNewAcl,ACL_REVISION,MAXDWORD,(LPVOID)pace,pace-
>Header.AceSize))
                  {
                        wprintf(L"  AddAce() failed, error %d\n",
GetLastError());
                        __leave;
                  }
                  else
                        wprintf(L"  AddAce() 2 is pretty fine!\n");

                  // set new dacl for the security descriptor
                  if (!SetSecurityDescriptorDacl(psdNew,TRUE,pNewAcl,FALSE))
                  {
                        wprintf(L"  SetSecurityDescriptorDacl() failed, error
%d\n", GetLastError());
                        __leave;
                  }
                  else
                        wprintf(L"  SetSecurityDescriptorDacl() is pretty
fine!\n");
```

```
                  // set the new security descriptor for the windowstation
                  if (!SetUserObjectSecurity(hwinsta, &si, psdNew))
                  {
                        wprintf(L"  SetUserObjectSecurity() failed, error
%d\n", GetLastError());
                        __leave;
                  }
                  else
                        wprintf(L"  SetUserObjectSecurity() is pretty
fine!\n");

                  // indicate success
                  bSuccess = TRUE;
                  }
                  __finally
                  {
                        // free the allocated buffers
                        wprintf(L"  Freeing up all the allocated
buffers...\n");
                        if (pace != NULL)
                              HeapFree(GetProcessHeap(), 0, (LPVOID)pace);
                        if (pNewAcl != NULL)
                              HeapFree(GetProcessHeap(), 0, (LPVOID)pNewAcl);
                        if (psd != NULL)
                              HeapFree(GetProcessHeap(), 0, (LPVOID)psd);
                        if (psdNew != NULL)
                              HeapFree(GetProcessHeap(), 0, (LPVOID)psdNew);
                  }
                  return bSuccess;
}

BOOL AddTheAceDesktop(HDESK hdesk, PSID psid)
{
      ACL_SIZE_INFORMATION aclSizeInfo;
      BOOL                 bDaclExist;
      BOOL                 bDaclPresent;
      BOOL                 bSuccess  = FALSE; // assume function will fail
      DWORD                dwNewAclSize;
      DWORD                dwSidSize = 0;
      DWORD                dwSdSizeNeeded;
      PACL                 pacl;
      PACL                 pNewAcl;
      PSECURITY_DESCRIPTOR psd       = NULL;
      PSECURITY_DESCRIPTOR psdNew    = NULL;
      PVOID                pTempAce;
      SECURITY_INFORMATION si        = DACL_SECURITY_INFORMATION;
      unsigned int         i;

      __try
      {
            // obtain the security descriptor for the desktop object
            if (!GetUserObjectSecurity(hdesk,&si,psd,dwSidSize,&dwSdSizeNeeded))
            {
                  if (GetLastError() == ERROR_INSUFFICIENT_BUFFER)
                  {
                        psd =
(PSECURITY_DESCRIPTOR)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwSdSizeNeeded
);
```

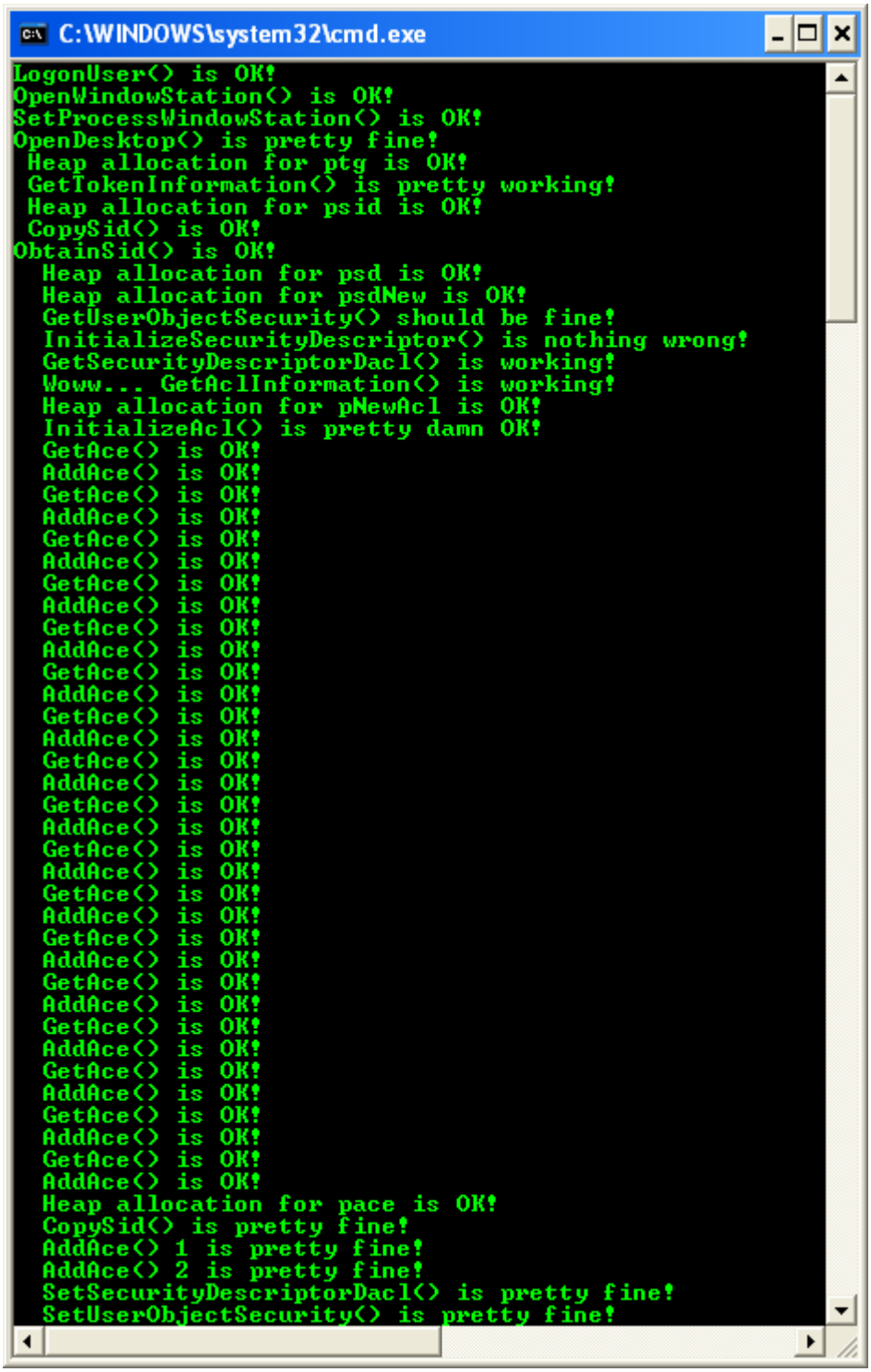
```
                        if (psd == NULL)
                              __leave;
                        else
                              wprintf(L"   Heap allocation for psd is OK!\n");

                        psdNew =
(PSECURITY_DESCRIPTOR)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwSdSizeNeeded
);

                        if (psdNew == NULL)
                              __leave;
                        else
                              wprintf(L"   Heap allocation for psdNew is
OK!\n");

                        dwSidSize = dwSdSizeNeeded;

                        if
(!GetUserObjectSecurity(hdesk,&si,psd,dwSidSize,&dwSdSizeNeeded))
                              __leave;
                        else
                              wprintf(L"   GetUserObjectSecurity() is OK!\n");
                  }
                  else
                        __leave;
            }
            else
                  wprintf(L"   GetUserObjectSecurity() is pretty fine!\n");

            // create a new security descriptor
            if
(!InitializeSecurityDescriptor(psdNew,SECURITY_DESCRIPTOR_REVISION))
            {
                  wprintf(L"   InitializeSecurityDescriptor() failed! error
%d\n", GetLastError());
                  __leave;
            }
            else
                  wprintf(L"   InitializeSecurityDescriptor() is pretty
fine!\n");

            // obtain the dacl from the security descriptor
            if (!GetSecurityDescriptorDacl(psd,&bDaclPresent,&pacl,&bDaclExist))
            {
                  wprintf(L"   GetSecurityDescriptorDacl() failed! error %d\n",
GetLastError());
                  __leave;
            }
            else
                  wprintf(L"   GetSecurityDescriptorDacl() is pretty fine!\n");

            // initialize
            ZeroMemory(&aclSizeInfo, sizeof(ACL_SIZE_INFORMATION));
            aclSizeInfo.AclBytesInUse = sizeof(ACL);

            // call only if NULL dacl
            if (pacl != NULL)
            {
```

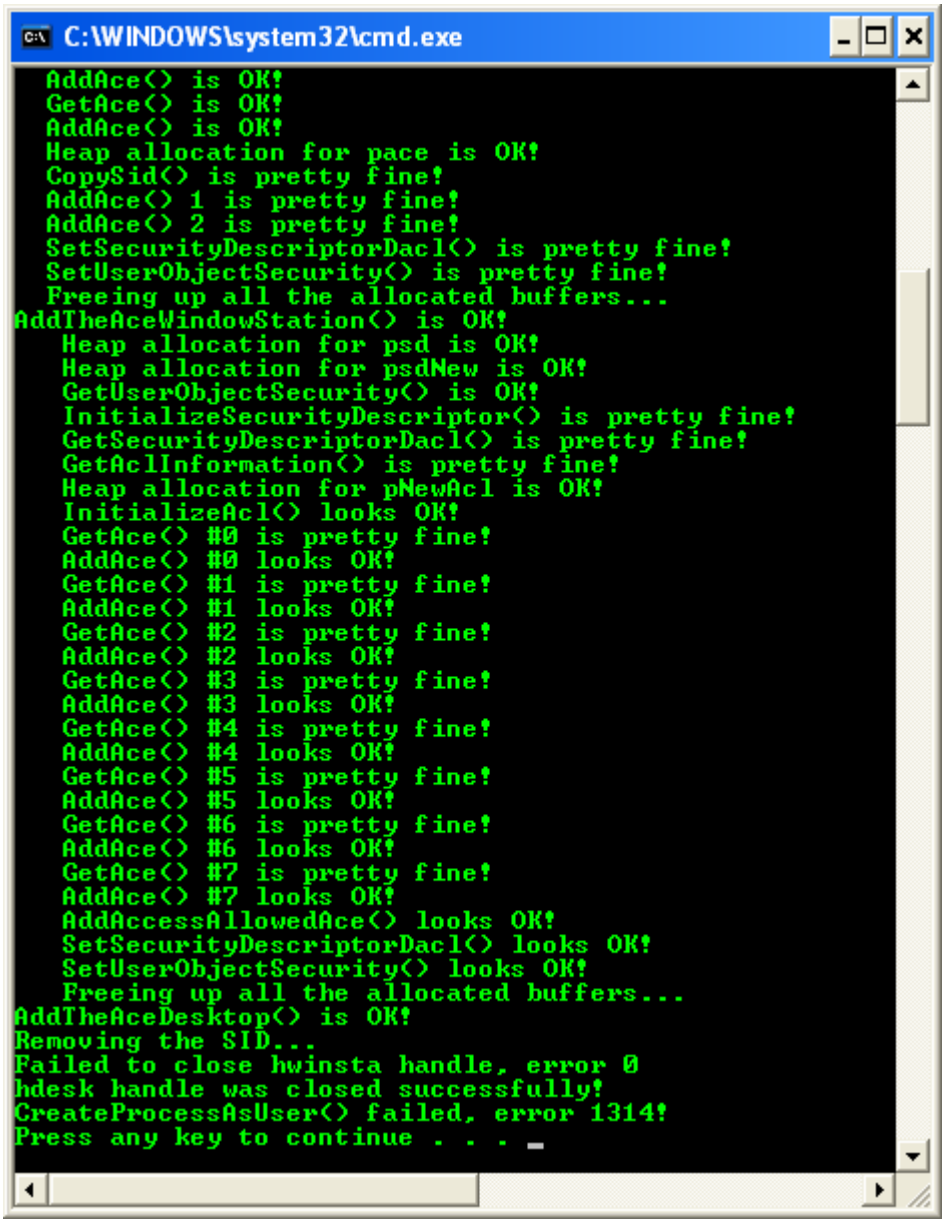
```
                  // determine the size of the ACL info
                  if
(!GetAclInformation(pacl,(LPVOID)&aclSizeInfo,sizeof(ACL_SIZE_INFORMATION),AclSi
zeInformation))
                  {
                        wprintf(L"   GetAclInformation() failed! error %d\n",
GetLastError());
                        __leave;
                  }
                  else
                        wprintf(L"   GetAclInformation() is pretty fine!\n");
            }

            // compute the size of the new acl
            dwNewAclSize = aclSizeInfo.AclBytesInUse +
sizeof(ACCESS_ALLOWED_ACE) + GetLengthSid(psid) - sizeof(DWORD);

            // allocate buffer for the new acl
            pNewAcl =
(PACL)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwNewAclSize);

            if (pNewAcl == NULL)
            {
                  wprintf(L"   Heap allocation for pNewAcl failed! error %d\n",
GetLastError());
                  __leave;
            }
            else
                  wprintf(L"   Heap allocation for pNewAcl is OK!\n");

            // initialize the new acl
            if (!InitializeAcl(pNewAcl, dwNewAclSize, ACL_REVISION))
            {
                  wprintf(L"   InitializeAcl() failed! error %d\n",
GetLastError());
                  __leave;
            }
            else
                  wprintf(L"   InitializeAcl() looks OK!\n");

            // if DACL is present, copy it to a new DACL
            if (bDaclPresent) // only copy if DACL was present
            {
                  // copy the ACEs to our new ACL
                  if (aclSizeInfo.AceCount)
                  {
                        for (i=0; i < aclSizeInfo.AceCount; i++)
                        {
                              // get an ACE
                              if (!GetAce(pacl, i, &pTempAce))
                              {
                                    wprintf(L"   GetAce() #%d failed! error
%d\n", i, GetLastError());
                                    __leave;
                              }
                              else
                                    wprintf(L"   GetAce() #%d is pretty
fine!\n", i);
```

```
                                    // add the ACE to the new ACL
                                    if (!AddAce(
                                          pNewAcl,
                                          ACL_REVISION,
                                          MAXDWORD,
                                          pTempAce,
                                          ((PACE_HEADER)pTempAce)->AceSize))
                                    {
                                          wprintf(L"   AddAce() #%d failed! error
%d\n", i, GetLastError());
                                          __leave;
                                    }
                                    else
                                          wprintf(L"   AddAce() #%d looks OK!\n", i);
                              }
                        }
                  }

                  // add ace to the dacl
                  if (!AddAccessAllowedAce(pNewAcl,ACL_REVISION,DESKTOP_ALL,psid))
                  {
                        wprintf(L"   AddAccessAllowedAce() failed! error %d\n",
GetLastError());
                        __leave;
                  }
                  else
                        wprintf(L"   AddAccessAllowedAce() looks OK!\n");

                  // set new dacl to the new security descriptor
                  if (!SetSecurityDescriptorDacl(psdNew,TRUE,pNewAcl,FALSE))
                  {
                        wprintf(L"   SetSecurityDescriptorDacl() failed! error %d\n",
GetLastError());
                        __leave;
                  }
                  else
                        wprintf(L"   SetSecurityDescriptorDacl() looks OK!\n");

                  // set the new security descriptor for the desktop object
                  if (!SetUserObjectSecurity(hdesk, &si, psdNew))
                  {
                        wprintf(L"   SetUserObjectSecurity() failed! error %d\n",
GetLastError());
                        __leave;
                  }
                  else
                        wprintf(L"   SetUserObjectSecurity() looks OK!\n");

                  // indicate success
                  bSuccess = TRUE;
            }
            __finally
            {
                  // free buffers
                  wprintf(L"   Freeing up all the allocated buffers...\n");
                  if (pNewAcl != NULL)
                        HeapFree(GetProcessHeap(), 0, (LPVOID)pNewAcl);
```

```
        if (psd != NULL)
              HeapFree(GetProcessHeap(), 0, (LPVOID)psd);
        if (psdNew != NULL)
              HeapFree(GetProcessHeap(), 0, (LPVOID)psdNew);
}
return bSuccess;
}
```

Build and run the project. The following screenshot is a sample output.


C:\WINDOWS\system32\cmd.exe
LogonUser() is OK!
OpenWindowStation() is OK!
SetProcessWindowStation() is OK!
OpenDesktop() is pretty fine!
Heap allocation for ptg is OK!
GetTokenInformation() is pretty working!
Heap allocation for psid is OK!
CopySid() is OK!
ObtainSid() is OK!
Heap allocation for psd is OK!
Heap allocation for psdNew is OK!
GetUserObjectSecurity() should be fine!
InitializeSecurityDescriptor() is nothing wrong!
GetSecurityDescriptorDacl() is working!
Woww... GetAclInformation() is working!
Heap allocation for pNewAcl is OK!
InitializeAcl() is pretty damn OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
GetAce() is OK!
AddAce() is OK!
Heap allocation for pace is OK!
CopySid() is pretty fine!
AddAce() 1 is pretty fine!
AddAce() 2 is pretty fine!
SetSecurityDescriptorDacl() is pretty fine!
SetUserObjectSecurity() is pretty fine!



Starting an Interactive Client Process in C++ Working Program Example

The following example uses the LogonUser() function to start a new logon session for a client. The example gets the logon SID from the client's access token, and uses it to add access control entries (ACEs) to the discretionary access control list (DACL) of the interactive window station and desktop. The ACEs allow the client access to the interactive desktop for the duration of the logon session. Next, the example calls the ImpersonateLoggedOnUser() function to ensure that it has access to the client's executable file. A call to the CreateProcessAsUser() function creates the client's process, specifying that it run in the interactive desktop. Note that your process must have the SE_ASSIGNPRIMARYTOKEN_NAME and SE_INCREASE_QUOTA_NAME privileges for successful execution of CreateProcessAsUser(). Before the function returns, it calls the RevertToSelf() function to end the caller's impersonation of the client.

Create a new empty Win32 console application project. Give a suitable project name and change the project location if needed.
Then, add the source file and give it a suitable name.
Next, add the following source code.

```
#include <windows.h>
#include <stdio.h>

// Constants
#define DESKTOP_ALL (DESKTOP_READOBJECTS | DESKTOP_CREATEWINDOW | \
DESKTOP_CREATEMENU | DESKTOP_HOOKCONTROL | DESKTOP_JOURNALRECORD | \
DESKTOP_JOURNALPLAYBACK | DESKTOP_ENUMERATE | DESKTOP_WRITEOBJECTS | \
DESKTOP_SWITCHDESKTOP | STANDARD_RIGHTS_REQUIRED)

#define WINSTA_ALL (WINSTA_ENUMDESKTOPS | WINSTA_READATTRIBUTES | \
WINSTA_ACCESSCLIPBOARD | WINSTA_CREATEDESKTOP | \
WINSTA_WRITEATTRIBUTES | WINSTA_ACCESSGLOBALATOMS | \
WINSTA_EXITWINDOWS | WINSTA_ENUMERATE | WINSTA_READSCREEN | \
STANDARD_RIGHTS_REQUIRED)

#define GENERIC_ACCESS (GENERIC_READ | GENERIC_WRITE | \
GENERIC_EXECUTE | GENERIC_ALL)

// Prototypes
BOOL GetLogonSID(HANDLE hToken, PSID *ppsid);
VOID FreeLogonSID (PSID *ppsid);
BOOL StartInteractiveClientProcess (
    LPTSTR lpszUsername,    // client to log on
    LPTSTR lpszDomain,      // domain of client's account
    LPTSTR lpszPassword,    // client's password
    LPTSTR lpCommandLine    // command line to execute
      );
BOOL AddAceToWindowStation(HWINSTA hwinsta, PSID psid);
BOOL AddAceToDesktop(HDESK hdesk, PSID psid);
BOOL RemoveAceFromWindowStation(HWINSTA hwinsta, PSID psid);
BOOL RemoveAceFromDesktop(HDESK hdesk, PSID psid);

int wmain(int argc, WCHAR **argv)
{
      // The user is a valid account on the local computer
      WCHAR lpszUsername[] = L"Johnny";   // Valid username
      WCHAR lpszDomain[] = L".";    // Using local user account database
      WCHAR lpszPassword[] = L"123";      // Johnny's password
      WCHAR lpCommandLine[] = L"C:\\Windows\\System32\\calc.exe";
      BOOL RetVal;

      // Call StartInteractiveClientProcess()
      RetVal = StartInteractiveClientProcess (
    lpszUsername,    // client to log on
    lpszDomain,      // domain of client's account
    lpszPassword,    // client's password
    lpCommandLine    // command line to execute
);
      wprintf(L"StartInteractiveClientProcess() returns %d\n", RetVal);
      return 0;
}
```

```
BOOL GetLogonSID(HANDLE hToken, PSID *ppsid)
{
	BOOL bSuccess = FALSE;
	DWORD dwIndex;
	DWORD dwLength = 0;
	PTOKEN_GROUPS ptg = NULL;

	wprintf(L"\nGetting the Logon SID...\n");

	// Verify the parameter passed in is not NULL
    if (ppsid == NULL)
	    goto Cleanup;
	else
		wprintf(L"ppsid is not NULL!\n");

	// Get required buffer size and allocate the TOKEN_GROUPS buffer
	if (!GetTokenInformation(
		hToken,		// handle to the access token
		TokenGroups,	// get information about the token's groups
		(LPVOID) ptg,	// pointer to TOKEN_GROUPS buffer
		0,		// size of buffer
		&dwLength	// receives required buffer size
		))
	{
		if (GetLastError() != ERROR_INSUFFICIENT_BUFFER)
		{
			wprintf(L"GetTokenInformation() 1 failed, error %d\n",
GetLastError());
			goto Cleanup;
		}
		else
			wprintf(L"Well, we have ample buffer...\n");

		ptg = (PTOKEN_GROUPS)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,
dwLength);

		if (ptg == NULL)
			goto Cleanup;
		else
			wprintf(L"Heap allocated for ptg!\n");
	}

	// Get the token group information from the access token.
	if (!GetTokenInformation(
		hToken,		// handle to the access token
		TokenGroups,	// get information about the token's groups
		(LPVOID)ptg,   // pointer to TOKEN_GROUPS buffer
		dwLength,	// size of buffer
		&dwLength	// receives required buffer size
		))
	{
		wprintf(L"GetTokenInformation() 2 failed, error %d\n",
GetLastError());
		goto Cleanup;
	}
	else
	{
```

```
            wprintf(L"GetTokenInformation() is pretty fine!\n");
            wprintf(L"ptg->GroupCount is %d\n", ptg->GroupCount);
            wprintf(L"ptg->Groups->Attributes is %d\n", ptg->Groups-
>Attributes);
      }

      // Loop through the groups to find the logon SID.
      for (dwIndex = 0; dwIndex < ptg->GroupCount; dwIndex++)
            if ((ptg->Groups[dwIndex].Attributes & SE_GROUP_LOGON_ID) ==
SE_GROUP_LOGON_ID)
            {
                  // Found the logon SID; make a copy of it.
                  dwLength = GetLengthSid(ptg->Groups[dwIndex].Sid);
                  *ppsid = (PSID) HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,
dwLength);

                  if (*ppsid == NULL)
                        goto Cleanup;
                  else
                        wprintf(L"Heap allocated for *ppsid!\n");

                  if (!CopySid(dwLength, *ppsid, ptg->Groups[dwIndex].Sid))
                  {
                        wprintf(L"CopySid() failed, error %d\n",
GetLastError());
                        HeapFree(GetProcessHeap(), 0, (LPVOID)*ppsid);
                        goto Cleanup;
                  }
                  else
                        wprintf(L"CopySid() is fine!\n");
                  break;
            }
            bSuccess = TRUE;

Cleanup:

            // Free the buffer for the token groups.
            if (ptg != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)ptg);

      return bSuccess;
}

VOID FreeLogonSID (PSID *ppsid)
{
      wprintf(L"\nFreeing up the Logon SID...\n\n");
      HeapFree(GetProcessHeap(), 0, (LPVOID)*ppsid);
}

BOOL StartInteractiveClientProcess(
                                                LPTSTR lpszUsername,    //
client to log on
                                                LPTSTR lpszDomain,      //
domain of client's account
                                                LPTSTR lpszPassword,    //
client's password
                                                LPTSTR lpCommandLine    //
command line to execute
```

```
                                                        )
{
        HANDLE      hToken;
        HDESK       hdesk = NULL;
        HWINSTA     hwinsta = NULL, hwinstaSave = NULL;
        PROCESS_INFORMATION pi;
        PSID pSid = NULL;
        STARTUPINFO si;
        BOOL bResult = FALSE;

        wprintf(L"Starting the interactive client process...\n");
        // Log the client on to the local computer.
        if
(!LogonUser(lpszUsername,lpszDomain,lpszPassword,LOGON32_LOGON_INTERACTIVE,LOGON
32_PROVIDER_DEFAULT,&hToken))
        {
                wprintf(L"LogonUser() failed, error %d\n", GetLastError());
                goto Cleanup;
        }
        else
                wprintf(L"LogonUser() is working!\n");

        // Save a handle to the caller's current window station.
        if ((hwinstaSave = GetProcessWindowStation()) == NULL)
                goto Cleanup;
        else
                wprintf(L"GetProcessWindowStation() should be fine!\n");

        // Get a handle to the interactive window station.
        hwinsta = OpenWindowStation(
                L"winsta0",                   // the interactive window station
                FALSE,                        // handle is not inheritable
                READ_CONTROL | WRITE_DAC);    // rights to read/write the DACL

        if (hwinsta == NULL)
                goto Cleanup;
        else
                wprintf(L"OpenWindowStation() is working!\n");

        // To get the correct default desktop, set the caller's
        // window station to the interactive window station.
        if (!SetProcessWindowStation(hwinsta))
                goto Cleanup;
        else
                wprintf(L"SetProcessWindowStation() 1 is working!\n");

        // Get a handle to the interactive desktop.
        hdesk = OpenDesktop(
                L"default",     // the interactive window station
                0,              // no interaction with other desktop processes
                FALSE,          // handle is not inheritable
                READ_CONTROL | // request the rights to read and write the DACL
                WRITE_DAC | DESKTOP_WRITEOBJECTS | DESKTOP_READOBJECTS);

        // Restore the caller's window station.
        if (!SetProcessWindowStation(hwinstaSave))
        {
```

```
            wprintf(L"SetProcessWindowStation() failed, error %d\n",
GetLastError());
            goto Cleanup;
      }
      else
            wprintf(L"SetProcessWindowStation() 2 is working!\n");

      if (hdesk == NULL)
            goto Cleanup;
      else
            wprintf(L"OpenDesktop() is working!\n");

      // Get the SID for the client's logon session.
      if (!GetLogonSID(hToken, &pSid))
      {
            wprintf(L"GetLogonSID() failed, error %d\n", GetLastError());
            goto Cleanup;
      }
      else
            wprintf(L"GetLogonSID() is working!\n");

      // Allow logon SID full access to interactive window station.
      if (!AddAceToWindowStation(hwinsta, pSid))
      {
            wprintf(L"AddAceToWindowStation() failed, error %d\n",
GetLastError());
            goto Cleanup;
      }
      else
            wprintf(L"AddAceToWindowStation() is working!\n");

      // Allow logon SID full access to interactive desktop.
      if (!AddAceToDesktop(hdesk, pSid))
      {
            wprintf(L"AddAceToDesktop() failed, error %d\n", GetLastError());
            goto Cleanup;
      }
      else
            wprintf(L"AddAceToDesktop() is working!\n");

      // Impersonate client to ensure access to executable file.
      if (!ImpersonateLoggedOnUser(hToken))
      {
            wprintf(L"ImpersonateLoggedOnUser() failed, error %d\n",
GetLastError());
            goto Cleanup;
      }
      else
            wprintf(L"ImpersonateLoggedOnUser() is working!\n");

      // Initialize the STARTUPINFO structure.
      // Specify that the process runs in the interactive desktop.
      ZeroMemory(&si, sizeof(STARTUPINFO));
      si.cb= sizeof(STARTUPINFO);
      si.lpDesktop = TEXT("winsta0\\default");

      // Launch the process in the client's logon session.
      bResult = CreateProcessAsUser(
```

```
            hToken,            // client's access token
            NULL,              // file to execute
            lpCommandLine,     // command line
            NULL,              // pointer to process SECURITY_ATTRIBUTES
            NULL,              // pointer to thread SECURITY_ATTRIBUTES
            FALSE,             // handles are not inheritable
            NORMAL_PRIORITY_CLASS | CREATE_NEW_CONSOLE,   // creation flags
            NULL,              // pointer to new environment block
            NULL,              // name of current directory
            &si,               // pointer to STARTUPINFO structure
            &pi                // receives information about new process
            );

      // When running many times, it will keeps adding more acls to the Windows
station until you hit some limit
      // http://support.microsoft.com/kb/185292/
      // Then SetUserObjectSecurity() will fails with ERROR_NOT_ENOUGH_QUOTA.
      // Undone any changes that were made to the Windows station and desktop.
      wprintf(L"Removing the ACE from Window station and desktop...\n");
      RemoveAceFromWindowStation(hwinsta, pSid);
      RemoveAceFromDesktop(hdesk, pSid);

      // End impersonation of client
      if(RevertToSelf() != 0)
            wprintf(L"RevertToSelf() is OK!\n");
      else
            wprintf(L"RevertToSelf() failed, error %d\n", GetLastError());

      if (bResult && pi.hProcess != INVALID_HANDLE_VALUE)
      {
            WaitForSingleObject(pi.hProcess, INFINITE);

            if(CloseHandle(pi.hProcess) != 0)
                  wprintf(L"pi.hProcess handle was closed successfully!\n");
            else
                  wprintf(L"Failed to close pi.hProcess handle, error %d\n",
GetLastError());
      }

      if (pi.hThread != INVALID_HANDLE_VALUE)
      {
            if(CloseHandle(pi.hThread) != 0)
                  wprintf(L"pi.hThread handle was closed successfully!\n");
            else
                  wprintf(L"Failed to close pi.hThread handle, error %d\n",
GetLastError());
      }
      else
            wprintf(L"pi.hThread handle already invalid!\n");

Cleanup:

      if (hwinstaSave != NULL)
            SetProcessWindowStation (hwinstaSave);
      // Free the buffer for the logon SID.
      if (pSid)
            FreeLogonSID(&pSid);
      // Close the handles to the interactive window station and desktop.
```

```
	if (hwinsta)
		CloseWindowStation(hwinsta);
	if (hdesk)
		CloseDesktop(hdesk);
	// Close the handle to the client's access token.
	if (hToken != INVALID_HANDLE_VALUE)
		CloseHandle(hToken);

	return bResult;
}

BOOL AddAceToWindowStation(HWINSTA hwinsta, PSID psid)
{
	ACCESS_ALLOWED_ACE   *pace;
	ACL_SIZE_INFORMATION aclSizeInfo;
	BOOL                 bDaclExist;
	BOOL                 bDaclPresent;
	BOOL                 bSuccess = FALSE;
	DWORD                dwNewAclSize;
	DWORD                dwSidSize = 0;
	DWORD                dwSdSizeNeeded;
	PACL                 pacl;
	PACL                 pNewAcl;
	PSECURITY_DESCRIPTOR psd = NULL;
	PSECURITY_DESCRIPTOR psdNew = NULL;
	PVOID                pTempAce;
	SECURITY_INFORMATION si = DACL_SECURITY_INFORMATION;
	unsigned int         i;

	wprintf(L"\nAdding ACE to WindowStation...\n");

	__try
	{
		// Obtain the DACL for the window station.
		if
(!GetUserObjectSecurity(hwinsta,&si,psd,dwSidSize,&dwSdSizeNeeded))
			if (GetLastError() == ERROR_INSUFFICIENT_BUFFER)
			{
				psd =
(PSECURITY_DESCRIPTOR)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwSdSizeNeeded
);

				if (psd == NULL)
					__leave;
				else
					wprintf(L"Heap allocated for psd!\n");

				psdNew =
(PSECURITY_DESCRIPTOR)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwSdSizeNeeded
);
				if (psdNew == NULL)
					__leave;
				else
					wprintf(L"Heap allocated for psdNew!\n");

				dwSidSize = dwSdSizeNeeded;
				if
(!GetUserObjectSecurity(hwinsta,&si,psd,dwSidSize,&dwSdSizeNeeded))
```

```
                        {
                              wprintf(L"GetUserObjectSecurity() failed, error
%d\n", GetLastError());
                              __leave;
                        }
                        else
                              wprintf(L"GetUserObjectSecurity() is working!\n");
                  }
                  else
                        __leave;

            // Create a new DACL.
            if
(!InitializeSecurityDescriptor(psdNew,SECURITY_DESCRIPTOR_REVISION))
            {
                  wprintf(L"InitializeSecurityDescriptor() failed, error %d\n",
GetLastError());
                  __leave;
            }
            else
                  wprintf(L"InitializeSecurityDescriptor() is working!\n");

            // Get the DACL from the security descriptor.
            if (!GetSecurityDescriptorDacl(psd,&bDaclPresent,&pacl,&bDaclExist))
            {
                  wprintf(L"GetSecurityDescriptorDacl() failed, error %d\n",
GetLastError());
                  __leave;
            }
            else
                  wprintf(L"GetSecurityDescriptorDacl() is working!\n");

            // Initialize the ACL
            SecureZeroMemory(&aclSizeInfo, sizeof(ACL_SIZE_INFORMATION));
            aclSizeInfo.AclBytesInUse = sizeof(ACL);

            // Call only if the DACL is not NULL
            if (pacl != NULL)
            {
                  // get the file ACL size info
                  if
(!GetAclInformation(pacl,(LPVOID)&aclSizeInfo,sizeof(ACL_SIZE_INFORMATION),AclSi
zeInformation))
                  {
                        wprintf(L"GetAclInformation() failed, error %d\n",
GetLastError());
                        __leave;
                  }
                  else
                        wprintf(L"GetAclInformation() is working!\n");
            }

            // Compute the size of the new ACL
            dwNewAclSize = aclSizeInfo.AclBytesInUse +
(2*sizeof(ACCESS_ALLOWED_ACE)) +
                  (2*GetLengthSid(psid)) - (2*sizeof(DWORD));

            // Allocate memory for the new ACL
```

```
            pNewAcl =
(PACL)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwNewAclSize);

            if (pNewAcl == NULL)
                  __leave;
            else
                  wprintf(L"Heap allocated for pNewAcl!\n");

            // Initialize the new DACL
            if (!InitializeAcl(pNewAcl, dwNewAclSize, ACL_REVISION))
            {
                  wprintf(L"InitializeAcl() failed, error %d\n",
GetLastError());
                  __leave;
            }
            else
                  wprintf(L"InitializeAcl() is working!\n");

            // If DACL is present, copy it to a new DACL
            if (bDaclPresent)
            {
                  // Copy the ACEs to the new ACL.
                  if (aclSizeInfo.AceCount)
                  {
                        for (i=0; i < aclSizeInfo.AceCount; i++)
                        {
                              // Get an ACE.
                              if (!GetAce(pacl, i, &pTempAce))
                              {
                                    wprintf(L"GetAce() failed, error %d\n",
GetLastError());
                                    __leave;
                              }
                              else
                                    wprintf(L"GetAce() is working!\n");

                              // Add the ACE to the new ACL.
                              if
(!AddAce(pNewAcl,ACL_REVISION,MAXDWORD,pTempAce,((PACE_HEADER)pTempAce)-
>AceSize))
                              {
                                    wprintf(L"AddAce() failed, error %d\n",
GetLastError());
                                    __leave;
                              }
                              else
                                    wprintf(L"AddAce() is working!\n");
                        }
                  }
            }

            // Add the first ACE to the window station
            pace = (ACCESS_ALLOWED_ACE
*)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,
                  sizeof(ACCESS_ALLOWED_ACE) + GetLengthSid(psid) -
sizeof(DWORD));

            if (pace == NULL)
```

```
                __leave;
        else
                wprintf(L"Heap allocated for pace!\n");

        pace->Header.AceType  = ACCESS_ALLOWED_ACE_TYPE;
        pace->Header.AceFlags = CONTAINER_INHERIT_ACE | INHERIT_ONLY_ACE |
OBJECT_INHERIT_ACE;
        pace->Header.AceSize  = (WORD)(sizeof(ACCESS_ALLOWED_ACE) +
GetLengthSid(psid) - sizeof(DWORD));
        pace->Mask            = GENERIC_ACCESS;

        if (!CopySid(GetLengthSid(psid), &pace->SidStart, psid))
        {
                wprintf(L"CopySid() failed, error %d\n", GetLastError());
                __leave;
        }
        else
                wprintf(L"CopySid() is working!\n");

        if (!AddAce(pNewAcl,ACL_REVISION,MAXDWORD,(LPVOID)pace,pace-
>Header.AceSize))
        {
                wprintf(L"AddAce() failed, error %d\n", GetLastError());
                __leave;
        }
        else
                wprintf(L"AddAce() 1 is working!\n");

        // Add the second ACE to the window station
        pace->Header.AceFlags = NO_PROPAGATE_INHERIT_ACE;
        pace->Mask            = WINSTA_ALL;

        if (!AddAce(pNewAcl,ACL_REVISION,MAXDWORD,(LPVOID)pace,pace-
>Header.AceSize))
        {
                wprintf(L"AddAce() failed, error %d\n", GetLastError());
                __leave;
        }
        else
                wprintf(L"AddAce() 2 is working!\n");

        // Set a new DACL for the security descriptor
        if (!SetSecurityDescriptorDacl(psdNew,TRUE,pNewAcl,FALSE))
        {
                wprintf(L"SetSecurityDescriptorDacl() failed, error %d\n",
GetLastError());
                __leave;
        }
        else
                wprintf(L"SetSecurityDescriptorDacl() is working!\n");

        // Set the new security descriptor for the window station
        if (!SetUserObjectSecurity(hwinsta, &si, psdNew))
        {
                wprintf(L"SetUserObjectSecurity() failed, error %d\n",
GetLastError());
                __leave;
        }
```

```
            else
                  wprintf(L"SetUserObjectSecurity() is working!\n");

            // Indicate success
            bSuccess = TRUE;
}
__finally
{
      // Free the allocated buffers
      if (pace != NULL)
            HeapFree(GetProcessHeap(), 0, (LPVOID)pace);
      if (pNewAcl != NULL)
            HeapFree(GetProcessHeap(), 0, (LPVOID)pNewAcl);
      if (psd != NULL)
            HeapFree(GetProcessHeap(), 0, (LPVOID)psd);
      if (psdNew != NULL)
            HeapFree(GetProcessHeap(), 0, (LPVOID)psdNew);
}
return bSuccess;
}

BOOL AddAceToDesktop(HDESK hdesk, PSID psid)
{
      ACL_SIZE_INFORMATION aclSizeInfo;
      BOOL                 bDaclExist;
      BOOL                 bDaclPresent;
      BOOL                 bSuccess = FALSE;
      DWORD                dwNewAclSize;
      DWORD                dwSidSize = 0;
      DWORD                dwSdSizeNeeded;
      PACL                 pacl;
      PACL                 pNewAcl;
      PSECURITY_DESCRIPTOR psd = NULL;
      PSECURITY_DESCRIPTOR psdNew = NULL;
      PVOID                pTempAce;
      SECURITY_INFORMATION si = DACL_SECURITY_INFORMATION;
      unsigned int         i;

      wprintf(L"\nAdding ACE to Desktop...\n");

      __try
      {
            // Obtain the security descriptor for the desktop object
            if (!GetUserObjectSecurity(hdesk,&si,psd,dwSidSize,&dwSdSizeNeeded))
            {
                  if (GetLastError() == ERROR_INSUFFICIENT_BUFFER)
                  {
                        psd =
(PSECURITY_DESCRIPTOR)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwSdSizeNeeded
);

                        if (psd == NULL)
                              __leave;

                        psdNew =
(PSECURITY_DESCRIPTOR)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwSdSizeNeeded
);
```

```
                        if (psdNew == NULL)
                              __leave;

                        dwSidSize = dwSdSizeNeeded;
                        if
(!GetUserObjectSecurity(hdesk,&si,psd,dwSidSize,&dwSdSizeNeeded))
                        {
                              wprintf(L"GetUserObjectSecurity() failed, error
%d\n", GetLastError());
                              __leave;
                        }
                  }
                  else
                        __leave;
            }

            // Create a new security descriptor
            if
(!InitializeSecurityDescriptor(psdNew,SECURITY_DESCRIPTOR_REVISION))
            {
                  wprintf(L"InitializeSecurityDescriptor() failed, error %d\n",
GetLastError());
                  __leave;
            }

            // Obtain the DACL from the security descriptor
            if (!GetSecurityDescriptorDacl(psd,&bDaclPresent,&pacl,&bDaclExist))
            {
                  wprintf(L"GetSecurityDescriptorDacl() failed, error %d\n",
GetLastError());
                  __leave;
            }

            // Initialize
            ZeroMemory(&aclSizeInfo, sizeof(ACL_SIZE_INFORMATION));
            aclSizeInfo.AclBytesInUse = sizeof(ACL);

            // Call only if NULL DACL
            if (pacl != NULL)
            {
                  // Determine the size of the ACL information
                  if
(!GetAclInformation(pacl,(LPVOID)&aclSizeInfo,sizeof(ACL_SIZE_INFORMATION),AclSi
zeInformation))
                  {
                        wprintf(L"GetAclInformation() failed, error %d\n",
GetLastError());
                        __leave;
                  }
            }

            // Compute the size of the new ACL
            dwNewAclSize = aclSizeInfo.AclBytesInUse +
sizeof(ACCESS_ALLOWED_ACE) +
                  GetLengthSid(psid) - sizeof(DWORD);

            // Allocate buffer for the new ACL
```

```
            pNewAcl =
(PACL)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwNewAclSize);
            if (pNewAcl == NULL)
                  __leave;
            // Initialize the new ACL
            if (!InitializeAcl(pNewAcl, dwNewAclSize, ACL_REVISION))
            {
                  wprintf(L"InitializeAcl() failed, error %d\n",
GetLastError());
                  __leave;
            }
            // If DACL is present, copy it to a new DACL
            if (bDaclPresent)
            {
                  // Copy the ACEs to the new ACL.
                  if (aclSizeInfo.AceCount)
                  {
                        for (i=0; i < aclSizeInfo.AceCount; i++)
                        {
                              // Get an ACE
                              if (!GetAce(pacl, i, &pTempAce))
                              {
                                    wprintf(L"GetAce() failed, error %d\n",
GetLastError());
                                    __leave;
                              }
                              // Add the ACE to the new ACL.
                              if
(!AddAce(pNewAcl,ACL_REVISION,MAXDWORD,pTempAce,((PACE_HEADER)pTempAce)-
>AceSize))
                              {
                                    wprintf(L"AddAce() failed, error %d\n",
GetLastError());
                                    __leave;
                              }
                        }
                  }
            }

            // Add ACE to the DACL
            if (!AddAccessAllowedAce(pNewAcl,ACL_REVISION,DESKTOP_ALL,psid))
            {
                  wprintf(L"AddAccessAllowedAce() failed, error %d\n",
GetLastError());
                  __leave;
            }
            // Set new DACL to the new security descriptor
            if (!SetSecurityDescriptorDacl(psdNew,TRUE,pNewAcl,FALSE))
            {
                  wprintf(L"SetSecurityDescriptorDacl() failed, error %d\n",
GetLastError());
                  __leave;
            }
            // Set the new security descriptor for the desktop object
            if (!SetUserObjectSecurity(hdesk, &si, psdNew))
            {
                  wprintf(L"SetUserObjectSecurity() failed, error %d\n",
GetLastError());
```

```
                  __leave;
            }
            // Indicate success
            bSuccess = TRUE;
      }
      __finally
      {
            // Free buffers
            if (pNewAcl != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)pNewAcl);
            if (psd != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)psd);
            if (psdNew != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)psdNew);
      }
      return bSuccess;
}

BOOL RemoveAceFromWindowStation(HWINSTA hwinsta, PSID psid)
{
      ACL_SIZE_INFORMATION aclSizeInfo;
      BOOL                 bDaclExist;
      BOOL                 bDaclPresent;
      BOOL                 bSuccess = FALSE;
      DWORD                dwNewAclSize;
      DWORD                dwSidSize = 0;
      DWORD                dwSdSizeNeeded;
      PACL                 pacl;
      PACL                 pNewAcl;
      PSECURITY_DESCRIPTOR psd = NULL;
      PSECURITY_DESCRIPTOR psdNew = NULL;
      ACCESS_ALLOWED_ACE*  pTempAce;
      SECURITY_INFORMATION si = DACL_SECURITY_INFORMATION;
      unsigned int         i;

      wprintf(L"\nRemoving ACE from Window Station...\n");
      __try
      {
            // Obtain the DACL for the window station
            if
(!GetUserObjectSecurity(hwinsta,&si,psd,dwSidSize,&dwSdSizeNeeded))
                  if (GetLastError() == ERROR_INSUFFICIENT_BUFFER)
                  {
                        psd =
(PSECURITY_DESCRIPTOR)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwSdSizeNeeded
);

                        if (psd == NULL)
                              __leave;
                        psdNew =
(PSECURITY_DESCRIPTOR)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwSdSizeNeeded
);
                        if (psdNew == NULL)
                              __leave;

                        dwSidSize = dwSdSizeNeeded;
                        if
(!GetUserObjectSecurity(hwinsta,&si,psd,dwSidSize,&dwSdSizeNeeded))
```

```
                        {
                              wprintf(L"GetUserObjectSecurity() failed, error
%d\n", GetLastError());
                              __leave;
                        }
                  }
                  else
                        __leave;

            // Create a new DACL
            if
(!InitializeSecurityDescriptor(psdNew,SECURITY_DESCRIPTOR_REVISION))
            {
                  wprintf(L"InitializeSecurityDescriptor() failed, error %d\n",
GetLastError());
                  __leave;
            }

            // Get the DACL from the security descriptor
            if (!GetSecurityDescriptorDacl(psd,&bDaclPresent,&pacl,&bDaclExist))
            {
                  wprintf(L"GetSecurityDescriptorDacl() failed, error %d\n",
GetLastError());
                  __leave;
            }

            // Initialize the ACL
            ZeroMemory(&aclSizeInfo, sizeof(ACL_SIZE_INFORMATION));
            aclSizeInfo.AclBytesInUse = sizeof(ACL);
            // Call only if the DACL is not NULL
            if (pacl != NULL)
            {
                  // get the file ACL size info
                  if
(!GetAclInformation(pacl,(LPVOID)&aclSizeInfo,sizeof(ACL_SIZE_INFORMATION),AclSi
zeInformation))
                  {
                        wprintf(L"GetAclInformation() failed, error %d\n",
GetLastError());
                        __leave;
                  }
            }

            // Compute the size of the new ACL
            dwNewAclSize = aclSizeInfo.AclBytesInUse +
(2*sizeof(ACCESS_ALLOWED_ACE)) +
                  (2*GetLengthSid(psid)) - (2*sizeof(DWORD));

            // Allocate memory for the new ACL
            pNewAcl =
(PACL)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwNewAclSize);

            if (pNewAcl == NULL)
                  __leave;
            // Initialize the new DACL
            if (!InitializeAcl(pNewAcl, dwNewAclSize, ACL_REVISION))
            {
```

```
                  wprintf(L"InitializeAcl() failed, error %d\n",
GetLastError());
                  __leave;
            }
            // If DACL is present, copy it to a new DACL
            if (bDaclPresent)
            {
                  // Copy the ACEs to the new ACL.
                  if (aclSizeInfo.AceCount)
                  {
                        for (i=0; i < aclSizeInfo.AceCount; i++)
                        {
                              // Get an ACE.
                              if (!GetAce(pacl, i,
reinterpret_cast<void**>(&pTempAce)))
                              {
                                    wprintf(L"GetAce() failed, error %d\n",
GetLastError());
                                    __leave;
                              }
                              else
                                    wprintf(L"GetAce() looks OK!\n");

                              if (!EqualSid(psid, &pTempAce->SidStart))
                              {
                                    // Add the ACE to the new ACL.
                                    if
(!AddAce(pNewAcl,ACL_REVISION,MAXDWORD,pTempAce,((PACE_HEADER)pTempAce)-
>AceSize))
                                    {
                                          wprintf(L"AddAce() failed, error
%d\n", GetLastError());
                                          __leave;
                                    }
                                    else
                                          wprintf(L"AddAce() looks OK!\n");
                              }
                              else
                                    wprintf(L"EqualSid() is pretty fine!\n");
                        }
                  }
            }

            if(pacl != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)pacl);
            // Set a new DACL for the security descriptor
            if (!SetSecurityDescriptorDacl(psdNew,TRUE,pNewAcl,FALSE))
            {
                  wprintf(L"SetSecurityDescriptorDacl() failed, error %d\n",
GetLastError());
                  __leave;
            }
            // Set the new security descriptor for the window station
            if (!SetUserObjectSecurity(hwinsta, &si, psdNew))
            {
                  wprintf(L"SetUserObjectSecurity() failed, error %d\n",
GetLastError());
                  __leave;
```

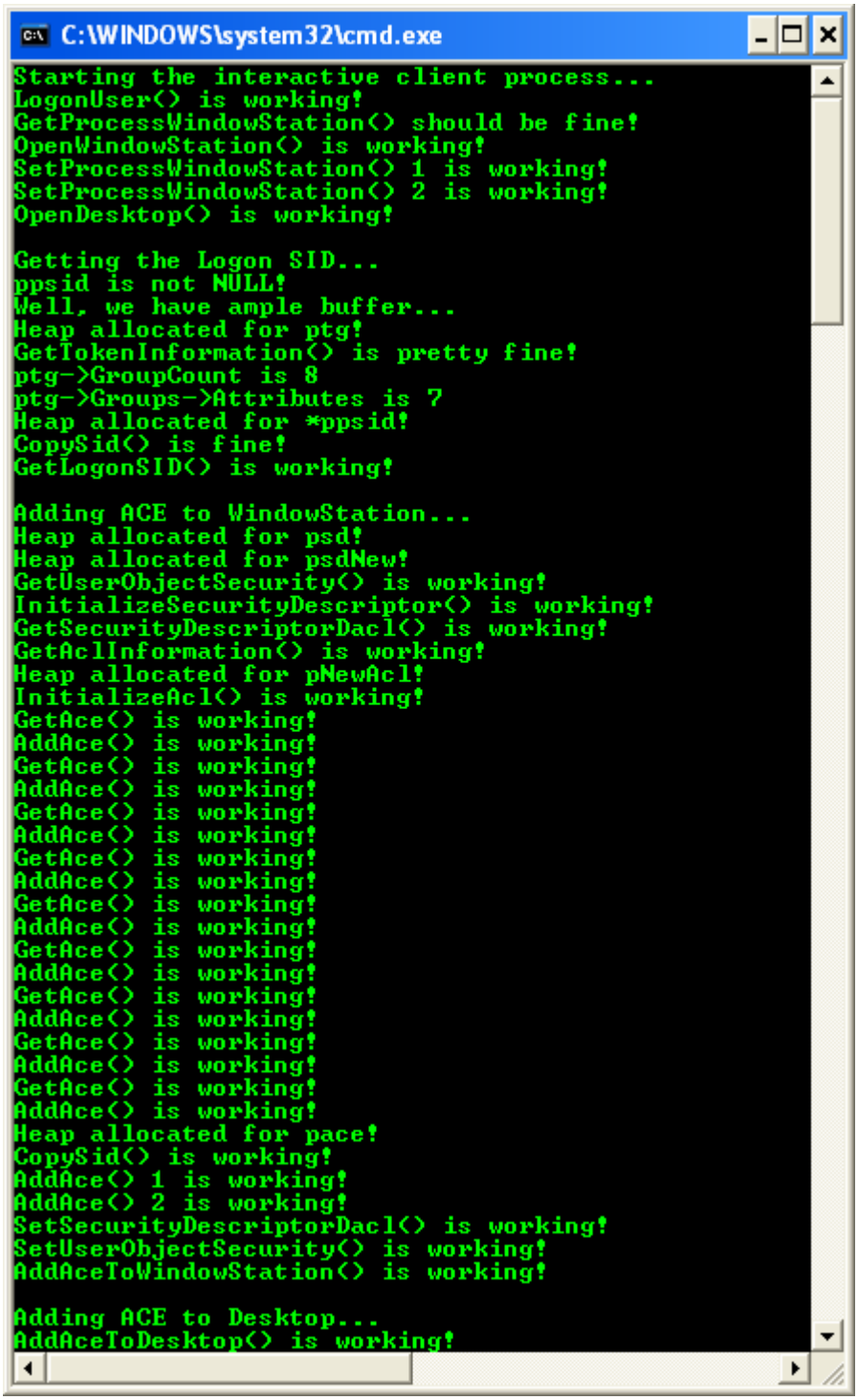
```
            }
            // Indicate success
            bSuccess = TRUE;
      }
      __finally
      {
            // Free the allocated buffers
            if(pacl != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)pacl);
            if (pNewAcl != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)pNewAcl);
            if (psd != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)psd);
            if (psdNew != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)psdNew);
      }
      return bSuccess;
}

BOOL RemoveAceFromDesktop(HDESK hdesk, PSID psid)
{
      ACL_SIZE_INFORMATION aclSizeInfo;
      BOOL                 bDaclExist;
      BOOL                 bDaclPresent;
      BOOL                 bSuccess = FALSE;
      DWORD                dwNewAclSize;
      DWORD                dwSidSize = 0;
      DWORD                dwSdSizeNeeded;
      PACL                 pacl;
      PACL                 pNewAcl;
      PSECURITY_DESCRIPTOR psd = NULL;
      PSECURITY_DESCRIPTOR psdNew = NULL;
      ACCESS_ALLOWED_ACE*  pTempAce;
      SECURITY_INFORMATION si = DACL_SECURITY_INFORMATION;
      unsigned int         i;

      wprintf(L"\nRemoving ACE from Desktop...\n");

      __try
      {
            // Obtain the security descriptor for the desktop object
            if (!GetUserObjectSecurity(hdesk,&si,psd,dwSidSize,&dwSdSizeNeeded))
            {
                  if (GetLastError() == ERROR_INSUFFICIENT_BUFFER)
                  {
                        psd =
(PSECURITY_DESCRIPTOR)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwSdSizeNeeded
);
                        if (psd == NULL)
                              __leave;
                        psdNew =
(PSECURITY_DESCRIPTOR)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwSdSizeNeeded
);
                        if (psdNew == NULL)
                              __leave;

                        dwSidSize = dwSdSizeNeeded;
```

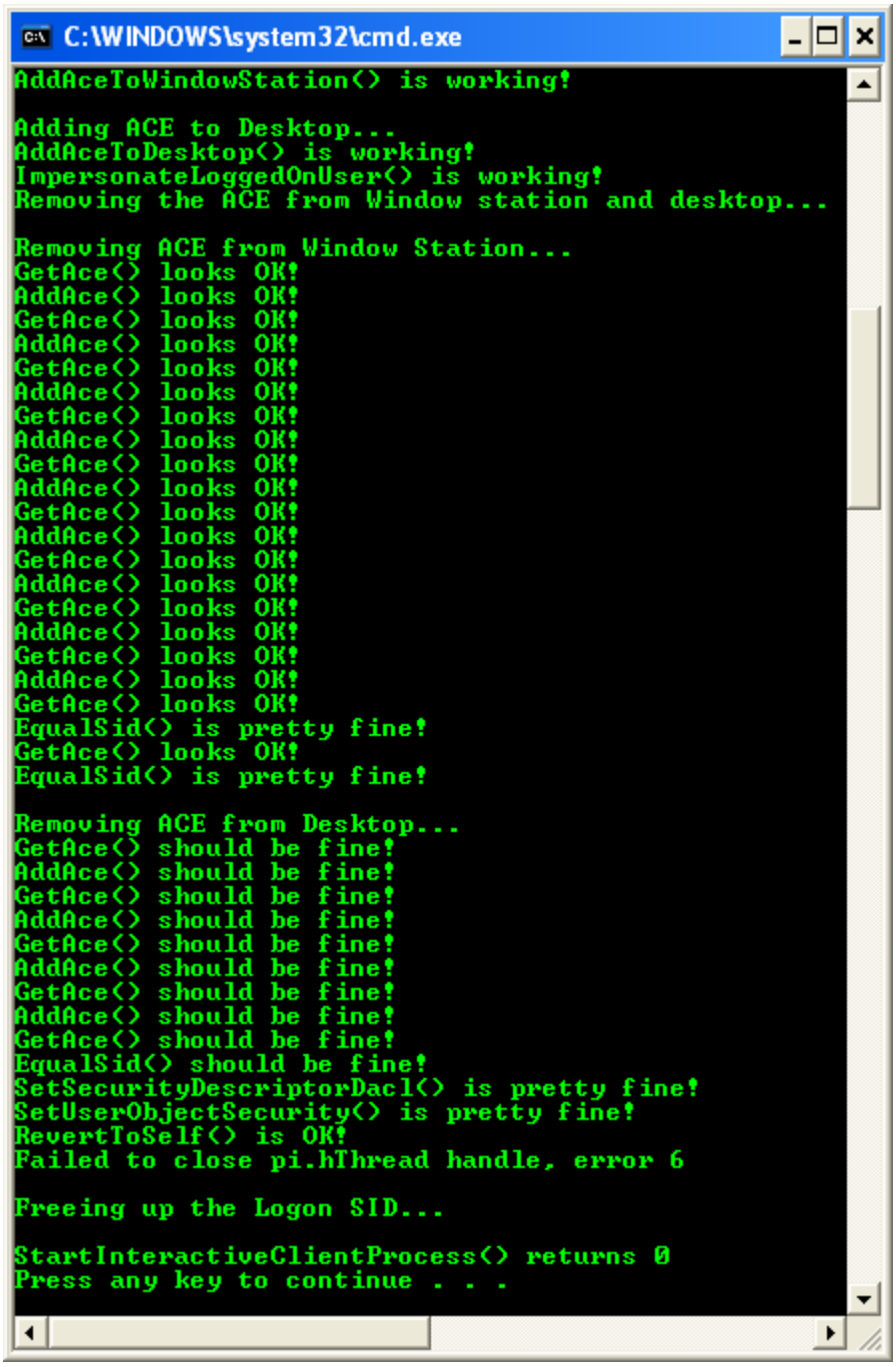
```
                        if
(!GetUserObjectSecurity(hdesk,&si,psd,dwSidSize,&dwSdSizeNeeded))
                        {
                              wprintf(L"GetUserObjectSecurity() failed, error
%d\n", GetLastError());
                              __leave;
                        }
                  }
                  else
                        __leave;
            }
            else
                  wprintf(L"GetUserObjectSecurity() should be OK!\n");

            // Create a new security descriptor
            if
(!InitializeSecurityDescriptor(psdNew,SECURITY_DESCRIPTOR_REVISION))
            {
                  wprintf(L"InitializeSecurityDescriptor() failed, error %d\n",
GetLastError());
                  __leave;
            }

            // Obtain the DACL from the security descriptor
            if (!GetSecurityDescriptorDacl(psd,&bDaclPresent,&pacl,&bDaclExist))
            {
                  wprintf(L"GetSecurityDescriptorDacl() failed, error %d\n",
GetLastError());
                  __leave;
            }

            // Initialize
            ZeroMemory(&aclSizeInfo, sizeof(ACL_SIZE_INFORMATION));
            aclSizeInfo.AclBytesInUse = sizeof(ACL);
            // Call only if NULL DACL
            if (pacl != NULL)
            {
                  // Determine the size of the ACL information
                  if
(!GetAclInformation(pacl,(LPVOID)&aclSizeInfo,sizeof(ACL_SIZE_INFORMATION),AclSi
zeInformation))
                  {
                        wprintf(L"GetAclInformation() failed, error %d\n",
GetLastError());
                        __leave;
                  }
            }

            // Compute the size of the new ACL
            dwNewAclSize = aclSizeInfo.AclBytesInUse +
sizeof(ACCESS_ALLOWED_ACE) + GetLengthSid(psid) - sizeof(DWORD);

            // Allocate buffer for the new ACL
            pNewAcl =
(PACL)HeapAlloc(GetProcessHeap(),HEAP_ZERO_MEMORY,dwNewAclSize);

            if (pNewAcl == NULL)
                  __leave;
```

```
            // Initialize the new ACL
            if (!InitializeAcl(pNewAcl, dwNewAclSize, ACL_REVISION))
            {
                  wprintf(L"InitializeAcl() failed, error %d\n",
GetLastError());
                  __leave;
            }
            // If DACL is present, copy it to a new DACL
            if (bDaclPresent)
            {
                  // Copy the ACEs to the new ACL.
                  if (aclSizeInfo.AceCount)
                  {
                        for (i=0; i < aclSizeInfo.AceCount; i++)
                        {
                              // Get an ACE.
                              if (!GetAce(pacl, i,
reinterpret_cast<void**>(&pTempAce)))
                              {
                                    wprintf(L"GetAce() failed, error %d\n",
GetLastError());
                                    __leave;
                              }
                              else
                                    wprintf(L"GetAce() should be fine!\n");

                              if (!EqualSid( psid, &pTempAce->SidStart))
                              {
                                    // Add the ACE to the new ACL.
                                    if
(!AddAce(pNewAcl,ACL_REVISION,MAXDWORD,pTempAce,((PACE_HEADER)pTempAce)-
>AceSize))
                                    {
                                          wprintf(L"AddAce() failed, error
%d\n", GetLastError());
                                          __leave;
                                    }
                                    else
                                          wprintf(L"AddAce() should be
fine!\n");
                              }
                              else
                                    wprintf(L"EqualSid() should be fine!\n");
                        }
                  }
            }

            // Set new DACL to the new security descriptor
            if (!SetSecurityDescriptorDacl(psdNew,TRUE,pNewAcl,FALSE))
            {
                  wprintf(L"SetSecurityDescriptorDacl() failed, error %d\n",
GetLastError());
                  __leave;
            }
            else
                  wprintf(L"SetSecurityDescriptorDacl() is pretty fine!\n");

            // Set the new security descriptor for the desktop object
```

```
            if (!SetUserObjectSecurity(hdesk, &si, psdNew))
            {
                  wprintf(L"SetUserObjectSecurity() failed, error %d\n",
GetLastError());
                  __leave;
            }
            else
                  wprintf(L"SetUserObjectSecurity() is pretty fine!\n");

            // Indicate success
            bSuccess = TRUE;
      }
      __finally
      {
            // Free buffers
            if(pacl != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)pacl);
            if (pNewAcl != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)pNewAcl);
            if (psd != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)psd);
            if (psdNew != NULL)
                  HeapFree(GetProcessHeap(), 0, (LPVOID)psdNew);
      }
      return bSuccess;
}
```

Build and run the project. The following screenshot is a sample output.


C:\WINDOWS\system32\cmd.exe
Starting the interactive client process...
LogonUser() is working!
GetProcessWindowStation() should be fine!
OpenWindowStation() is working!
SetProcessWindowStation() 1 is working!
SetProcessWindowStation() 2 is working!
OpenDesktop() is working!
Getting the Logon SID...
ppsid is not NULL!
Well, we have ample buffer...
Heap allocated for ptg!
GetTokenInformation() is pretty fine!
ptg->GroupCount is 8
ptg->Groups->Attributes is 7
Heap allocated for *ppsid!
CopySid() is fine!
GetLogonSID() is working!
Adding ACE to WindowStation...
Heap allocated for psd!
Heap allocated for psdNew!
GetUserObjectSecurity() is working!
InitializeSecurityDescriptor() is working!
GetSecurityDescriptorDacl() is working!
GetAclInformation() is working!
Heap allocated for pNewAcl!
InitializeAcl() is working!
GetAce() is working!
AddAce() is working!
GetAce() is working!
AddAce() is working!
GetAce() is working!
AddAce() is working!
GetAce() is working!
AddAce() is working!
GetAce() is working!
AddAce() is working!
GetAce() is working!
AddAce() is working!
GetAce() is working!
AddAce() is working!
GetAce() is working!
AddAce() is working!
GetAce() is working!
AddAce() is working!
Heap allocated for pace!
CopySid() is working!
AddAce() 1 is working!
AddAce() 2 is working!
SetSecurityDescriptorDacl() is working!
SetUserObjectSecurity() is working!
AddAceToWindowStation() is working!
Adding ACE to Desktop...
AddAceToDesktop() is working!


C:\WINDOWS\system32\cmd.exe
AddAceToWindowStation() is working!
Adding ACE to Desktop...
AddAceToDesktop() is working!
ImpersonateLoggedOnUser() is working!
Removing the ACE from Window station and desktop...
Removing ACE from Window Station...
GetAce() looks OK!
AddAce() looks OK!
GetAce() looks OK!
AddAce() looks OK!
GetAce() looks OK!
AddAce() looks OK!
GetAce() looks OK!
AddAce() looks OK!
GetAce() looks OK!
AddAce() looks OK!
GetAce() looks OK!
AddAce() looks OK!
GetAce() looks OK!
AddAce() looks OK!
GetAce() looks OK!
AddAce() looks OK!
GetAce() looks OK!
AddAce() looks OK!
GetAce() looks OK!
EqualSid() is pretty fine!
GetAce() looks OK!
EqualSid() is pretty fine!
Removing ACE from Desktop...
GetAce() should be fine!
AddAce() should be fine!
GetAce() should be fine!
AddAce() should be fine!
GetAce() should be fine!
AddAce() should be fine!
GetAce() should be fine!
AddAce() should be fine!
GetAce() should be fine!
EqualSid() should be fine!
SetSecurityDescriptorDacl() is pretty fine!
SetUserObjectSecurity() is pretty fine!
RevertToSelf() is OK!
Failed to close pi.hThread handle, error 6
Freeing up the Logon SID...
StartInteractiveClientProcess() returns 0
Press any key to continue . . .