1. Enabling and Disabling Privileges program example
2. Privilege and SACL Program Example
3. Searching for a SID in an Access Token program example 1
4. Searching for a SID in an Access Token Program Example 2
5. Getting the Logon (Session) SID in C++

Enabling and Disabling Privileges Code Snippet Example

The last program example in the previous Windows User & Groups, we failed to add the SACL to the ACL because we don't have privilege to do that task. We already demonstrated how to enable a privilege in our earlier program example. Again, in the following program example we will try to enable the required privilege to accomplish our task. Enabling a privilege in an access token allows the process to perform system-level actions that it could not previously. Your application should thoroughly verify that the privilege is appropriate to the type of account, especially for the following powerful privileges:

Privilege constant/string	Display name
SE_ASSIGNPRIMARYTOKEN_NAME SeAssignPrimaryTokenPrivilege	Replace a process level token
SE_BACKUP_NAME SeBackupPrivilege	Backup files and directories
SE_DEBUG_NAME SeDebugPrivilege	Debug programs
SE_INCREASE_QUOTA_NAME SeIncreaseQuotaPrivilege	Adjust memory quotas for a process
SE_TCB_NAME SeTchPrivilege	Act as part of the operating system

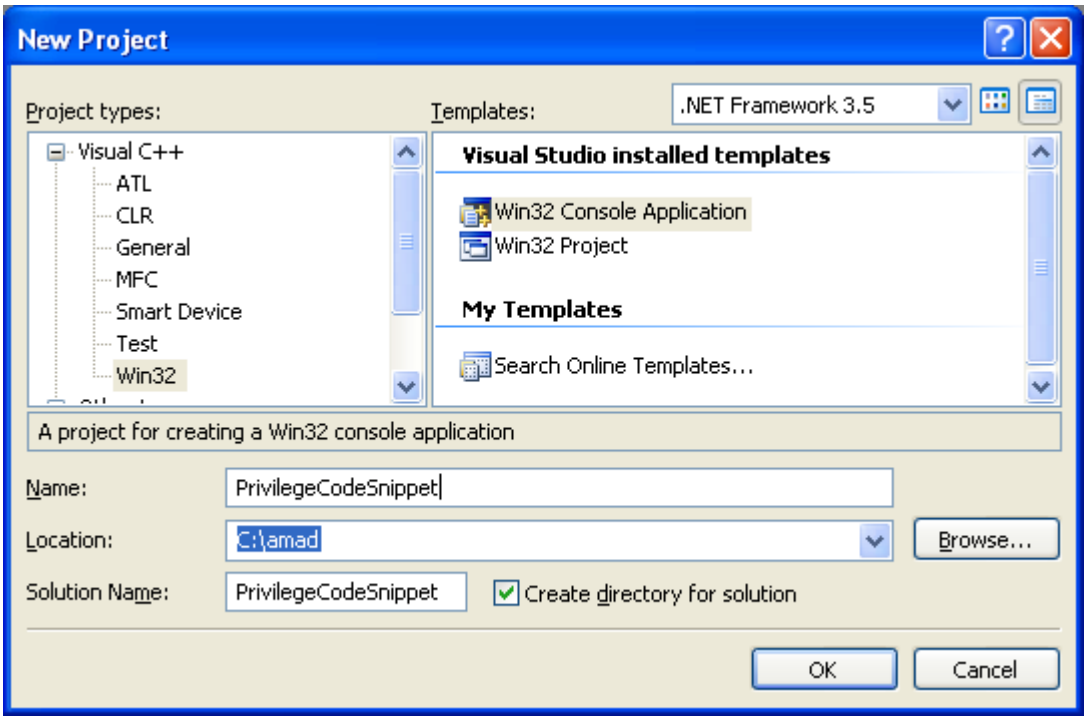
Table 1

Before enabling any of these potentially dangerous privileges, determine that functions or operations in your code actually require the privileges. For example, very few functions in the operating system actually require the SeTchPrivilege. The following example shows how to enable or disable a privilege in an access token. The example calls the LookupPrivilegeValue() function to get the LUID that the local system uses to identify the privilege. Then the example calls the AdjustTokenPrivileges() function, which either enables or disables the privilege that

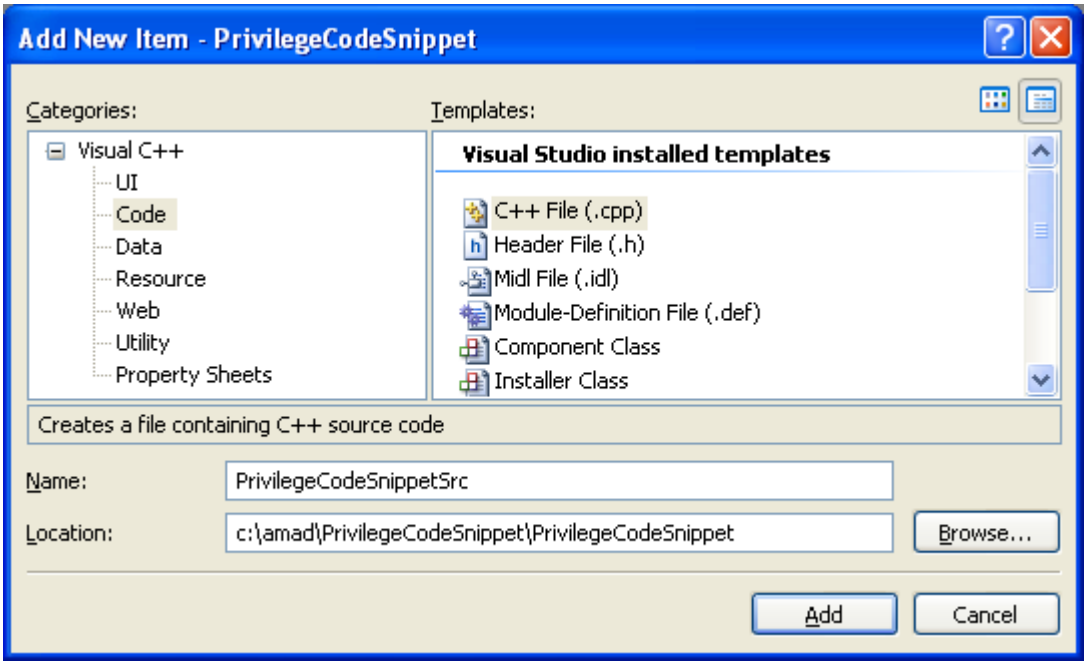
depends on the value of the bEnablePrivilege parameter. The following is a code portion used to enable/disable privilege.

```
BOOL SetPrivilege(
    HANDLE hToken,                  // access token handle
    LPCTSTR lpszPrivilege,    // name of privilege to enable/disable
    BOOL bEnablePrivilege     // to enable or disable privilege
    )
{
      TOKEN_PRIVILEGES tp;
      // Used by local system to identify the privilege
      LUID luid;

      if(!LookupPrivilegeValue(
            NULL,             // lookup privilege on local system
        lpszPrivilege,  // privilege to lookup
        &luid))               // receives LUID of privilege
      {
            wprintf(L"LookupPrivilegeValue() failed, error %u\n",
GetLastError());
            return FALSE;
      }

      tp.PrivilegeCount = 1;
      tp.Privileges[0].Luid = luid;

      // If TRUE, enable
      if(bEnablePrivilege)
            tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
      // else, disable
      else
            tp.Privileges[0].Attributes = 0;

      // Enable the privilege or disable all privileges
      if(!AdjustTokenPrivileges(
            hToken,
            FALSE,
            &tp,
            sizeof(TOKEN_PRIVILEGES),
            (PTOKEN_PRIVILEGES) NULL,
            (PDWORD) NULL))
      {
            wprintf(L"AdjustTokenPrivileges() failed, error %u\n",
GetLastError());
            return FALSE;
      }
      return TRUE;
}
```

The following is a working program example that enables/disables privilege.
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.

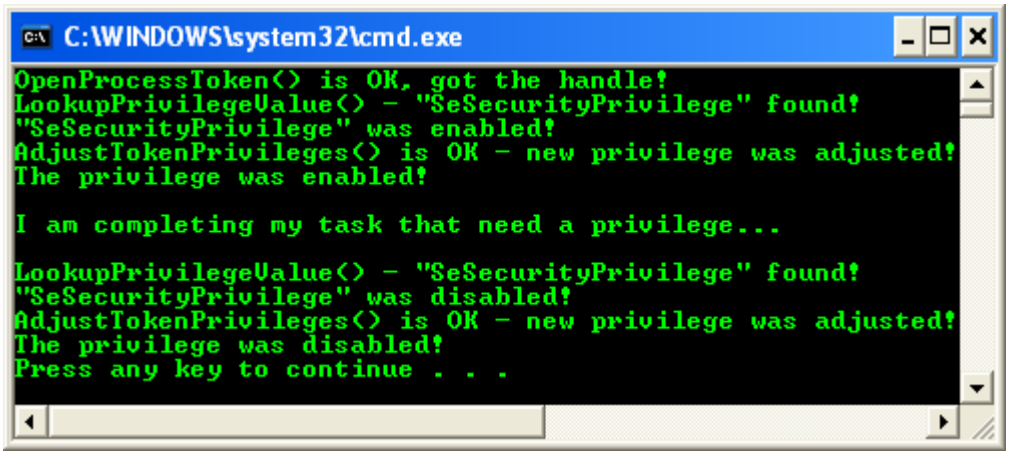
```
#include <windows.h>
#include <stdio.h>
```

```
// Enable/disable privilege routine
BOOL SetPrivilege(
                          HANDLE hToken,  // access token handle
                          LPCTSTR lpszPrivilege,    // name of privilege to
enable/disable
                          BOOL bEnablePrivilege    // to enable (or disable
privilege)
                          )
{
      // Token privilege structure
      TOKEN_PRIVILEGES tp;
      // Used by local system to identify the privilege
      LUID luid;

      if(!LookupPrivilegeValue(
            NULL,                  // lookup privilege on local system
            lpszPrivilege,         // privilege to lookup
            &luid))                      // receives LUID of privilege
      {
            wprintf(L"LookupPrivilegeValue() failed, error: %u\n",
GetLastError());
            return FALSE;
      }
      else
            wprintf(L"LookupPrivilegeValue() - \"%s\" found!\n",
lpszPrivilege);

      // Number of privilege
      tp.PrivilegeCount = 1;
      // Assign luid to the 1st count
      tp.Privileges[0].Luid = luid;

      // Enable/disable
      if(bEnablePrivilege)
      {
            // Enable
            tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
            wprintf(L"\"%s\" was enabled!\n", lpszPrivilege);
      }
      else
      {
            // Disable
            tp.Privileges[0].Attributes = 0;
            wprintf(L"\"%s\" was disabled!\n", lpszPrivilege);
      }

      // Adjusting the new privilege
      if(!AdjustTokenPrivileges(
            hToken,
            FALSE,      // If TRUE, function disables all privileges,
                        // if FALSE the function modifies privilege based on
the tp
            &tp,
            sizeof(TOKEN_PRIVILEGES),
            (PTOKEN_PRIVILEGES) NULL,
            (PDWORD) NULL))
      {
```

```
            wprintf(L"AdjustTokenPrivileges() failed to adjust the new
privilege, error: %u\n", GetLastError());
            return FALSE;
      }
      else
      {
            printf("AdjustTokenPrivileges() is OK - new privilege was
adjusted!\n");
      }
      return TRUE;
}

int wmain(int argc, WCHAR **argv)
{
      // The privilege to be adjusted
      LPCTSTR lpszPrivilege = L"SeSecurityPrivilege";
      // Change this BOOL value to set/unset the SE_PRIVILEGE_ENABLED
attribute
      // Initially to enable
      BOOL bEnablePrivilege = TRUE;
      HANDLE hToken;
      BOOL bRetVal;

      // Open a handle to the access token for the calling process
      if(!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES,
&hToken))
      {
            wprintf(L"OpenProcessToken() failed, error %u\n",
GetLastError());
            return FALSE;
      }
      else
            wprintf(L"OpenProcessToken() is OK, got the handle!\n");

      // Call the user defined SetPrivilege() function to
      // enable and set the needed privilege
      bRetVal = SetPrivilege(hToken, lpszPrivilege, bEnablePrivilege);

      if(!bRetVal)
      {
            wprintf(L"Failed to enable privilege, error %u\n",
GetLastError());
            return FALSE;
      }
      else
            wprintf(L"The privilege was enabled!\n");

      //************************************************
      // TODO: Complete your task which need the privilege
      wprintf(L"\nI am completing my task that need a privilege...\n\n");
      //************************************************

      // After we have completed our task, don't forget to disable the
privilege
      bEnablePrivilege = FALSE;
      bRetVal = SetPrivilege(hToken, lpszPrivilege, bEnablePrivilege);
      if(!bRetVal)
```

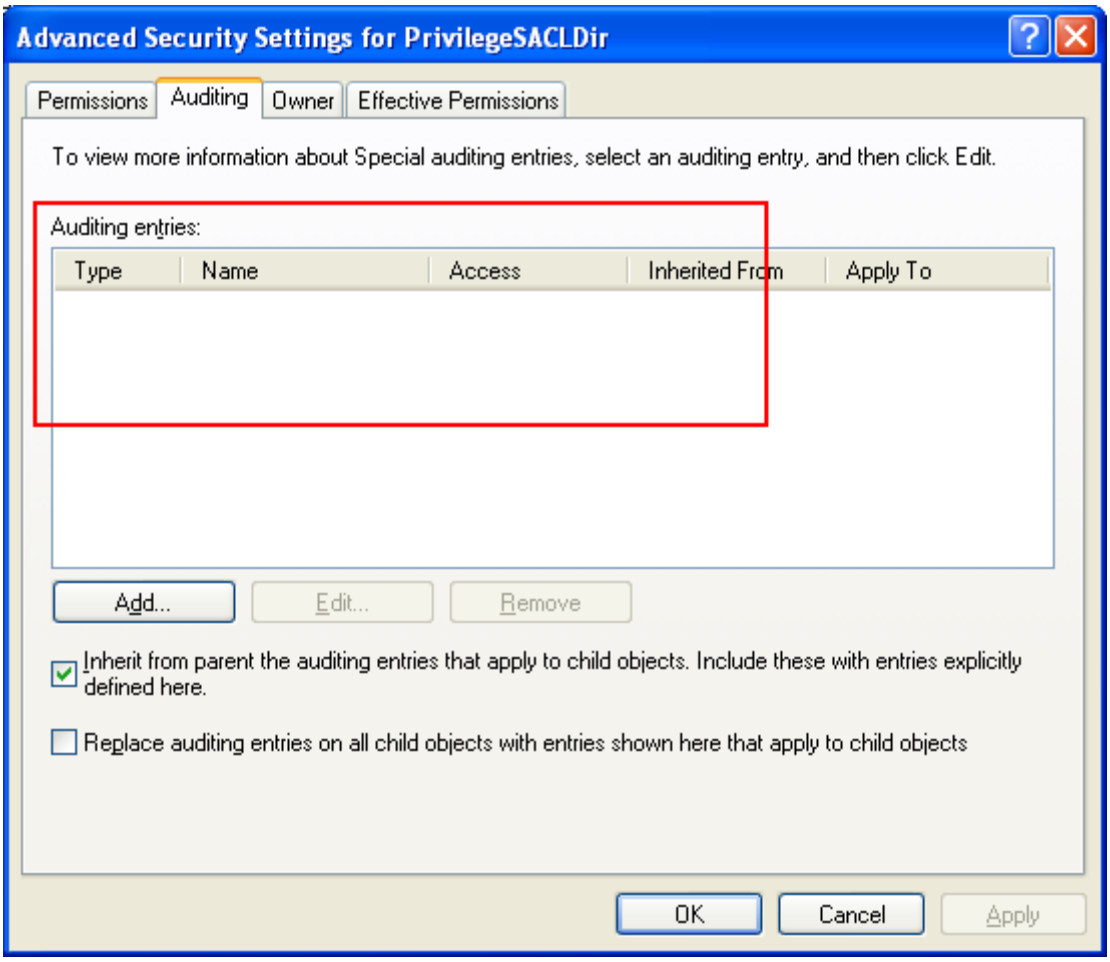
```
	{
		wprintf(L"Failed to disable the privilege, error %u\n",
GetLastError());
		return FALSE;
	}
	else
		wprintf(L"The privilege was disabled!\n");

	return 0;
}
```

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

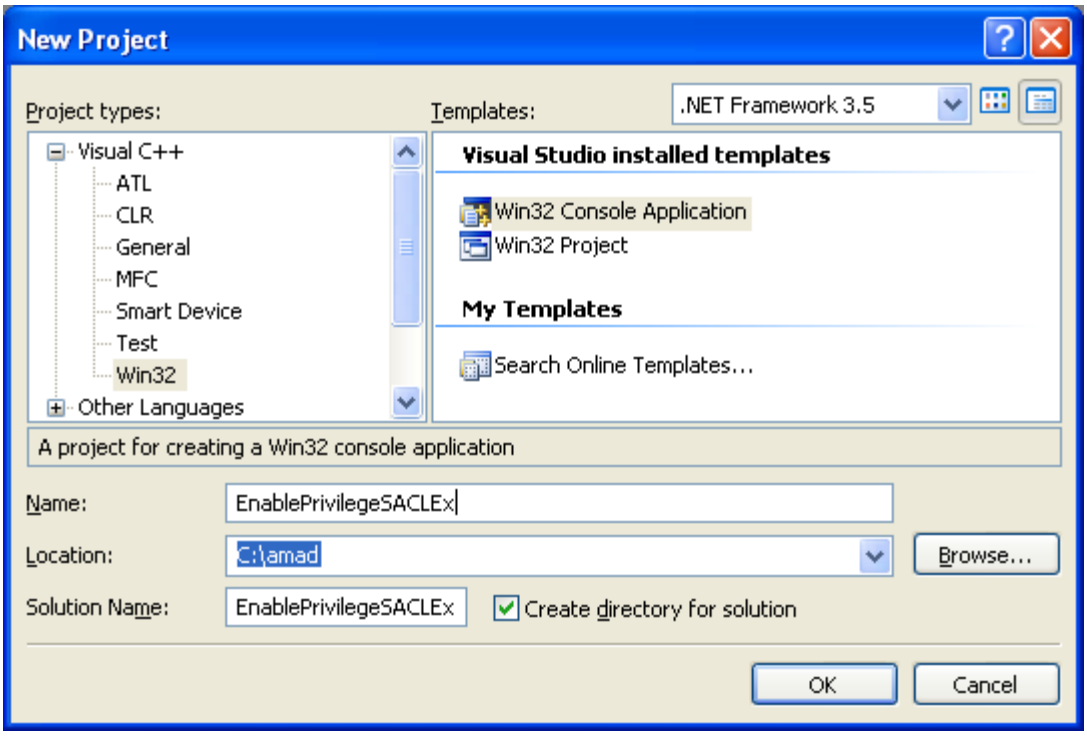


Privilege and SACL Program Example

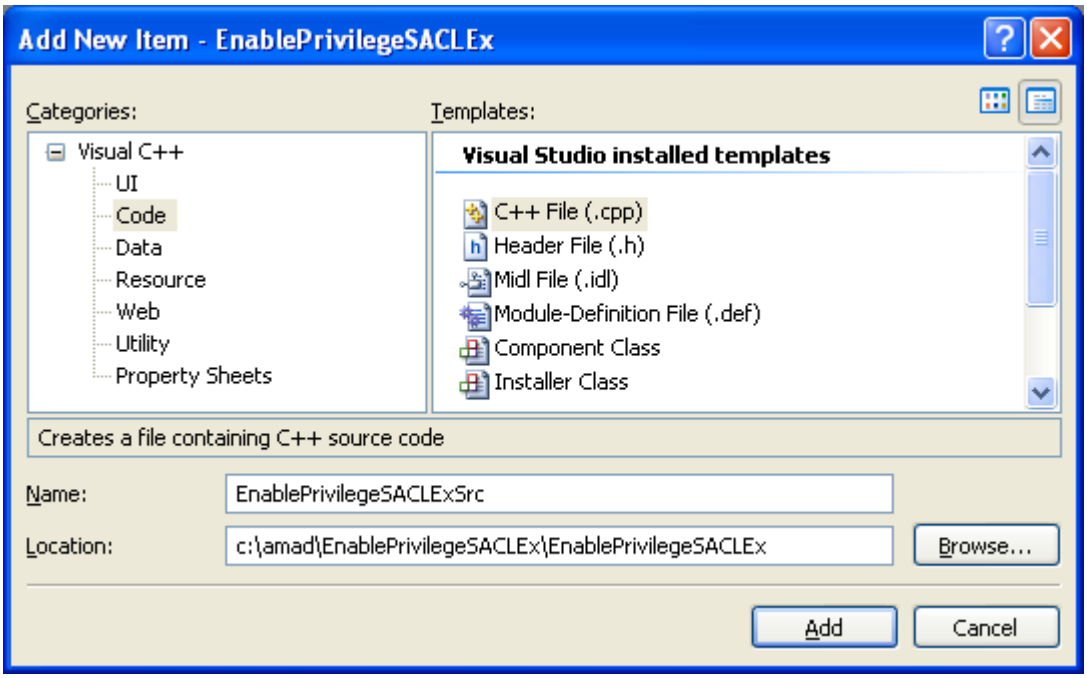
In the following example we will try to enable a privilege that needed in order to set the SACL using the same program example. First of all let create the PrivilegeSACLDir directory and verify the Auditing from property page (should be none).



Then execute the following program which is a modified previously created program.
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.

```
// In this program we are going to
// add Allow standard right access and set an ACE for SACL.
```

```
// This Win XP machine is logged in by user named Mike spoon
// who is a member of Administrators group...
// To access a SACL using the GetNamedSecurityInfo()
// or SetNamedSecurityInfo() functions, you have to enable
// the SE_SECURITY_NAME privilege
#include <windows.h>
#include <aclapi.h>
#include <stdio.h>

//********* Enabling/Disabling the privilege *********
BOOL SetPrivilege(
                        HANDLE hToken,              // access token handle
                        LPCTSTR lpszPrivilege,      // name of privilege to
enable/disable
                        BOOL bEnablePrivilege // to enable (or disable
privilege)
                        )
{
      TOKEN_PRIVILEGES tp;
      // Used by local system to identify the privilege
      LUID luid;

      if(!LookupPrivilegeValue(
            NULL,             // lookup privilege on local system
            lpszPrivilege,    // privilege to lookup
            &luid))                 // receives LUID of privilege
      {
            wprintf(L"LookupPrivilegeValue() failed to find %s privilege,
error: %u\n", lpszPrivilege, GetLastError());
            return FALSE;
      }
      else
            wprintf(L"LookupPrivilegeValue() is OK - Privilege \"%s\" was
found!\n", lpszPrivilege);

      // Set the privilege count
      tp.PrivilegeCount = 1;
      // Assign luid to the first count
      tp.Privileges[0].Luid = luid;

      // If TRUE
      if(bEnablePrivilege)
      {
            tp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;
            wprintf(L"\"%s\" privilege successfully enabled!\n",
lpszPrivilege);
      }
      else
      {
            tp.Privileges[0].Attributes = 0;
            wprintf(L"\"%s\" privilege successfully disabled!\n",
lpszPrivilege);
      }

      // Adjust to the new privilege
      if(!AdjustTokenPrivileges(
            hToken,
```

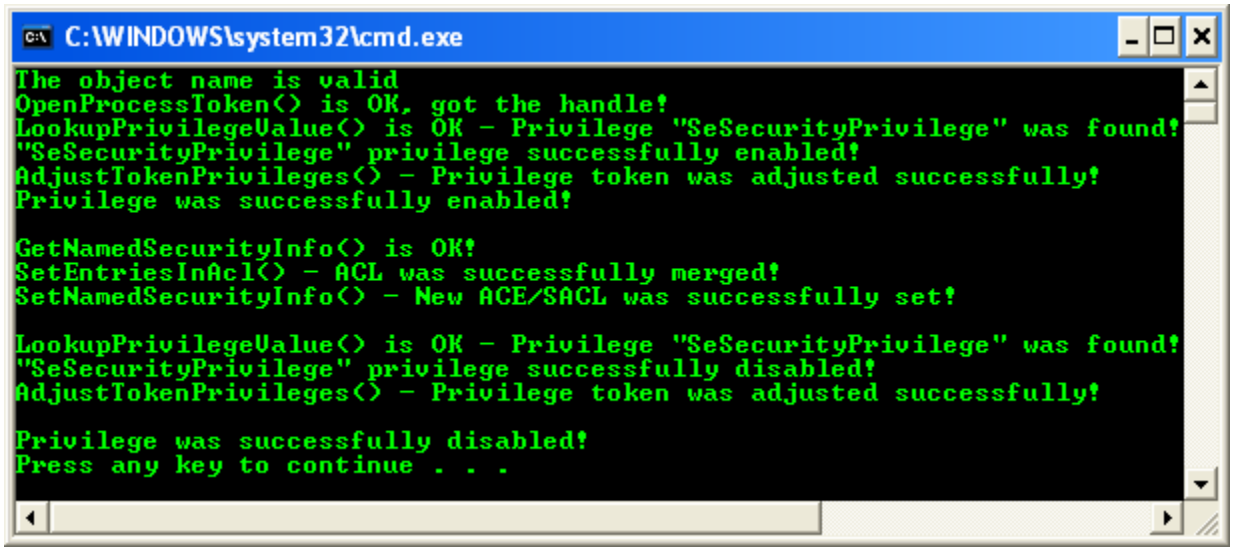
```
            FALSE,      // If TRUE, function disables all privileges,
                        // if FALSE the function modifies privileges based on
the tp
            &tp,
            sizeof(TOKEN_PRIVILEGES),
            (PTOKEN_PRIVILEGES) NULL,
            (PDWORD) NULL))
      {
            wprintf(L"AdjustTokenPrivileges() failed, error: %u\n",
GetLastError());
            return FALSE;
      }
      else
      {
            wprintf(L"AdjustTokenPrivileges() - Privilege token was adjusted
successfully!\n");
      }

      return TRUE;
}

//************** Clean up routine ***************
void Cleanup(PSECURITY_DESCRIPTOR pSS, PACL pNewSACL)
{

      if(pSS != NULL)
            LocalFree((HLOCAL) pSS);
      else
            wprintf(L"pSS was freed-up\n");

      if(pNewSACL != NULL)
            LocalFree((HLOCAL) pNewSACL);
      else
            wprintf(L"pNewSACL was freed-up\n");
}

//******** The main() **********
int main(int argc, WCHAR **argv)
{
      // Name of object, here we will add an ACE for a directory
      LPTSTR pszObjName = L"C:\\PrivilegeSACLDir";
      // Ttype of object, file or directory, a directory
      SE_OBJECT_TYPE ObjectType = SE_FILE_OBJECT;
      // Access mask for new ACE equal to 0X11000000 - GENERIC_ALL and
ACCESS_SYSTEM_SECURITY
      DWORD dwAccessRights = 0X11000000;
      // ACCESS_MODE AccessMode = SET_AUDIT_FAILURE;
      // Type of ACE, set audit for success
      ACCESS_MODE AccessMode = SET_AUDIT_SUCCESS;
      // Inheritance flags for new ACE. The OBJECT_INHERIT_ACE and
CONTAINER_INHERIT_ACE flags
      // are not propagated to an inherited ACE.
      DWORD dwInheritance = NO_PROPAGATE_INHERIT_ACE;
      // Format of trustee structure, the trustee is name
      TRUSTEE_FORM TrusteeForm = TRUSTEE_IS_NAME;
      // The needed privilege
      LPCTSTR lpszPrivilege = L"SeSecurityPrivilege";
```

```
	// The new trustee for the ACE is set to Mike spoon,
	// a normal user of Administrators group
	LPTSTR pszTrustee = L"Mike spoon";
	// Result
	DWORD dwRes = 0;
	// Existing and new SACL pointers...
	PACL pOldSACL = NULL, pNewSACL = NULL;
	// Security descriptor
	PSECURITY_DESCRIPTOR pSS = NULL;
	// EXPLICIT_ACCESS structure
	EXPLICIT_ACCESS ea;
	// Change this BOOL value to set/unset the SE_PRIVILEGE_ENABLED
attribute
	BOOL bEnablePrivilege = TRUE;
	// Handle to the running process that is this (running) program
	HANDLE hToken;
	BOOL bRetVal = FALSE;


	// Verify the object name validity
	if(pszObjName == NULL)
	{
		wprintf(L"The object name is invalid, error %u\n",
GetLastError());
		return ERROR_INVALID_PARAMETER;
	}
	else
		wprintf(L"The object name is valid\n");

	//*************** Get the handle to the process ******************
	// Open a handle to the access token for the calling process
	if(!OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES,
&hToken))
	{
		wprintf(L"OpenProcessToken() failed, error %u\n",
GetLastError());
		return 1;
	}
	else
		wprintf(L"OpenProcessToken() is OK, got the handle!\n");

	//******* Enabling privilege ********
	// Call the user defined SetPrivilege() function to enable privilege
	bRetVal = SetPrivilege(hToken, lpszPrivilege, bEnablePrivilege);
	// Verify
	if(!bRetVal)
	{
		wprintf(L"Failed to enable the privilege, error %u\n\n",
GetLastError());
		return 1;
	}
	else
		wprintf(L"Privilege was successfully enabled!\n\n");
	//******* End enabling privilege ********

	// By assuming that we have enabled the required privilege,
	// accomplish our task. Here, get a pointer to the existing SACL
```

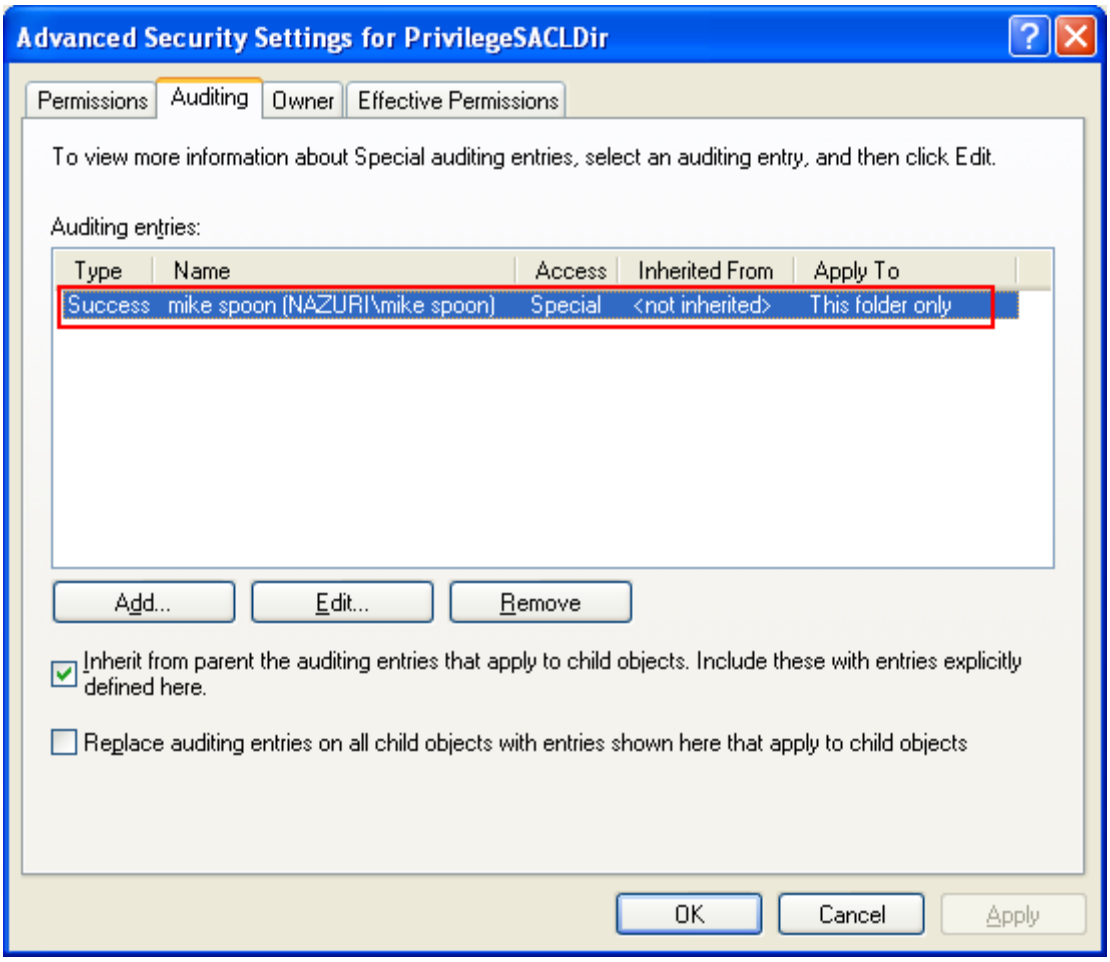
```
	dwRes = GetNamedSecurityInfo(pszObjName,
		ObjectType,
		SACL_SECURITY_INFORMATION,
		NULL, NULL, NULL,
		&pOldSACL, &pSS);

	// Verify
	if(dwRes != ERROR_SUCCESS)
	{
		wprintf(L"GetNamedSecurityInfo() failed, error %u\n", dwRes);
		Cleanup(pSS, pNewSACL);
		return 1;
	}
	else
		wprintf(L"GetNamedSecurityInfo() is OK!\n");

	// Initialize an EXPLICIT_ACCESS structure for the new ACE.
	// For more ACE entries, declare an array of the EXPLICIT_ACCESS
structure
	SecureZeroMemory(&ea, sizeof(EXPLICIT_ACCESS));
	ea.grfAccessPermissions = dwAccessRights;
	ea.grfAccessMode = AccessMode;
	ea.grfInheritance= dwInheritance;
	ea.Trustee.TrusteeForm = TrusteeForm;

	// Other structure elements that might be needed...
	// ea.Trustee.TrusteeType = TRUSTEE_IS_GROUP;
	// ea.Trustee.TrusteeType = TRUSTEE_IS_USER;

	// The trustee is Mike spoon
	ea.Trustee.ptstrName = pszTrustee;
	// Create a new ACL that merges the new ACE into the existing ACL.
	dwRes = SetEntriesInAcl(1, &ea, pOldSACL, &pNewSACL);
	if(dwRes != ERROR_SUCCESS)
	{
		wprintf(L"SetEntriesInAcl() failed, error %u\n", dwRes);
		Cleanup(pSS, pNewSACL);
		return 1;
	}
	else
		wprintf(L"SetEntriesInAcl() - ACL was successfully merged!\n");

	// Attach the new ACL as the object's SACL.
	dwRes = SetNamedSecurityInfo(pszObjName,
		ObjectType,
		SACL_SECURITY_INFORMATION,
		NULL, NULL, NULL,
		pNewSACL);

	if(dwRes != ERROR_SUCCESS)
	{
		wprintf(L"SetNamedSecurityInfo() failed, error %u\n", dwRes);
		Cleanup(pSS, pNewSACL);
		return 1;
	}
	else
```

```
            wprintf(L"SetNamedSecurityInfo() - New ACE/SACL was successfully
set!\n\n");

      // Disable the privilege
      //****************** Disabling privilege ******************
      bEnablePrivilege = FALSE;
      bRetVal = SetPrivilege(hToken, lpszPrivilege, bEnablePrivilege);
      // Verify
      if(!bRetVal)
      {
            wprintf(L"\nFailed to disable the privilege, error %u\n",
GetLastError());
            return 1;
      }
      else
            wprintf(L"\nPrivilege was successfully disabled!\n");
      //****************** End disabling privilege ***************

      // Close the process handle by calling the CloseHandle(hToken)
      CloseHandle(hToken);

      return 0;
}
```

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

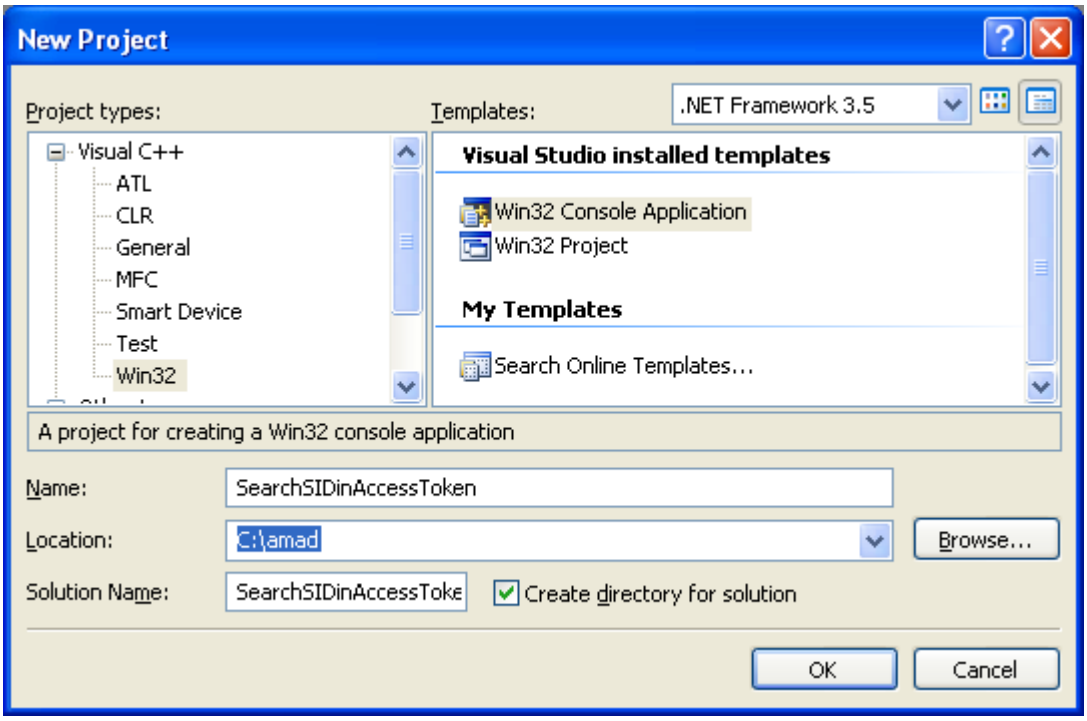


Well, it seems that we have enabled the privilege successfully and set the SACL. Let verify through the PrivilegeSACLDir directory property, the Advanced Security Settings property page.

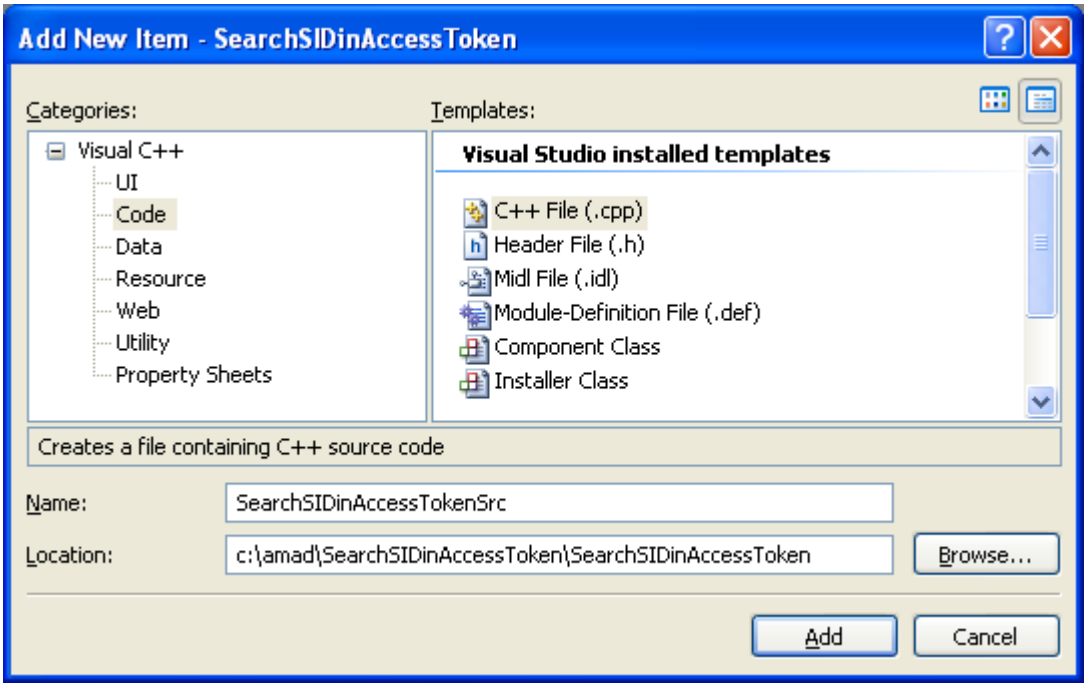


Searching for a SID in an Access Token Program Example 1

The following example uses the OpenProcessToken() and GetTokenInformation() functions to get the group memberships in an access token. Then it uses the AllocateAndInitializeSid() function to create a SID that identifies the well-known SID of the administrator group for the local computer. Next, it uses the EqualSid() function to compare the well-known SID with the group SIDs from the access token. If the SID is present in the token, the function checks the attributes of the SID to determine whether it is enabled.
The CheckTokenMembership() function used to determine whether a specified SID is present and enabled in an access token. This function eliminates potential misinterpretations of the active group membership if changes to access tokens are made in future releases.
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.

```
// Searching for a SID in an Access Token.
// This program run on the standalone Windows Xp Pro
```

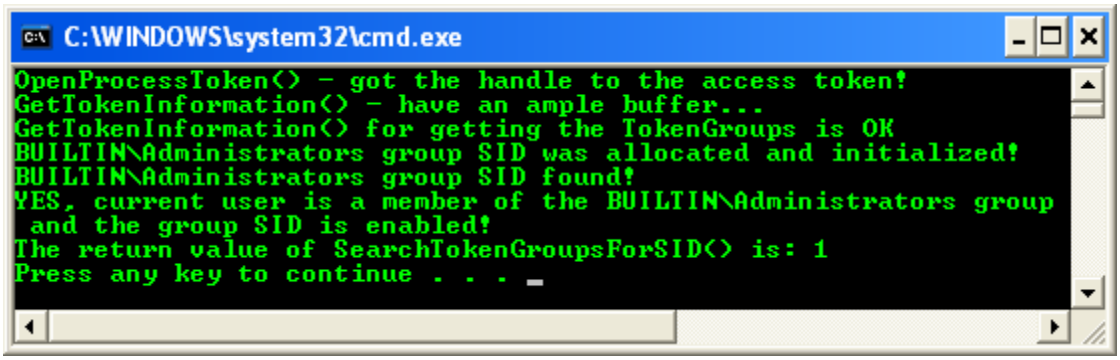
```
#include <windows.h>
#include <stdio.h>

// A constant
#define MAX_NAME 256

BOOL SearchTokenGroupsForSID(void)
{
	DWORD i, dwSize = 0, dwResult = 0;
	HANDLE hToken;
	PTOKEN_GROUPS pGroupInfo;
	SID_NAME_USE SidType;
	WCHAR lpName[MAX_NAME];
	WCHAR lpDomain[MAX_NAME];
	BYTE sidBuffer[100];
	PSID pSID = (PSID)&sidBuffer;

	// Open a handle to the access token for the calling process,
	// that is, this running program (process).  So we get the handle
	// to the token for the current user that run this program and/or login
to this machine.
	//  Depend on your task, change the TOKEN_QUERY to others
accordingly...
	if(!OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken))
	{
		wprintf(L"OpenProcessToken() failed, error %u\n",
GetLastError());
		return FALSE;
	}
	else
		wprintf(L"OpenProcessToken() - got the handle to the access
token!\n");

	// By assuming that we got the handle to the token...
	// Call GetTokenInformation() to get the buffer size for storage.
	// This is for Tokengroups, change accordingly for others such
	// as TokenUser, TokenOwner, TokenType etc.
	if(!GetTokenInformation(hToken, TokenGroups, NULL, dwSize, &dwSize))
	{
		dwResult = GetLastError();
		if(dwResult != ERROR_INSUFFICIENT_BUFFER)
		{
			wprintf(L"GetTokenInformation() failed, error %u\n",
dwResult);
			return FALSE;
		}
		else
			wprintf(L"GetTokenInformation() - have an ample
buffer...\n");
	}
	else
		wprintf(L"GetTokenInformation() - buffer for Token group is
OK\n");

	// By assuming that we got the storage, then allocate the buffer.
	pGroupInfo = (PTOKEN_GROUPS)GlobalAlloc(GPTR, dwSize);
	// Call GetTokenInformation() again to get the group information.
```

```
	if(!GetTokenInformation(hToken, TokenGroups, pGroupInfo, dwSize,
&dwSize))
	{
		wprintf(L"GetTokenInformation() failed, error %u\n",
GetLastError());
		return FALSE;
	}
	else
		wprintf(L"GetTokenInformation() for getting the TokenGroups is
OK\n");

	//****************** Playing with SIDs ***************************
	// Create a SID for the BUILTIN\Administrators group...
	// You can try other groups also lol as commented out on the following
	// codes. Uncomment/comment out the SIDs for testing....
	// This is 32 bit RID value. Applications that require longer RID
values,
	// use CreateWellKnownSid() instead
	//*********************** Administrator group **********************
	SID_IDENTIFIER_AUTHORITY SIDAuth = SECURITY_NT_AUTHORITY;
	if(!AllocateAndInitializeSid(&SIDAuth, 2,
		SECURITY_BUILTIN_DOMAIN_RID,
		DOMAIN_ALIAS_RID_ADMINS,
		0, 0, 0, 0, 0, 0,
		&pSID))
	{
		wprintf(L"AllocateAndInitializeSid() failed, error %u\n",
GetLastError());
		return FALSE;
	}
	else
		wprintf(L"BUILTIN\\Administrators group SID was allocated and
initialized!\n");

	//************************ Local group *******************
	// An example for creating a SID for the Local group...
	// SID_IDENTIFIER_AUTHORITY SIDAuth = SECURITY_LOCAL_SID_AUTHORITY;
	//
	// if(!AllocateAndInitializeSid(&SIDAuth, 1,
	//                 SECURITY_LOCAL_RID,
	//                 0, 0, 0, 0, 0, 0, 0,
	//                 &pSID))
	// {
	//    wprintf(L"AllocateAndInitializeSid() error %u\n",
GetLastError());
	//    return FALSE;
	// }
	// else
	//          wprintf(L"AllocateAndInitializeSid(), SID for Local group
is\n successfully created\n");
	//********************* Authenticated users **********************
	// Another example for creating a SID for the Authenticated users...
	// SID_IDENTIFIER_AUTHORITY SIDAuth = SECURITY_NT_AUTHORITY;
	// if(!AllocateAndInitializeSid(&SIDAuth, 1,
	//                 SECURITY_AUTHENTICATED_USER_RID,
	//                 0, 0, 0, 0, 0, 0, 0,
	//                 &pSID))
```

```
	// {
	// wprintf(L"AllocateAndInitializeSid() error %u\n", GetLastError());
	// return FALSE;
	// }
	// else
	// wprintf(L"AllocateAndInitializeSid(), SID for Local group is\n
successfully created\n");
	//*************************************************************

	// Loop through the group SIDs looking for the created group SID.
	for(i=0; i<pGroupInfo->GroupCount; i++)
	{
		// Compare the created SID with the available group SIDs
		if(EqualSid(pSID, pGroupInfo->Groups[i].Sid))
		{
			// Lookup the account name and print it.
			dwSize = MAX_NAME;
			if(!LookupAccountSid(NULL,
				pGroupInfo->Groups[i].Sid,
				lpName,
				&dwSize,
				lpDomain,
				&dwSize,
				&SidType))
			{
				// If not found or something wrong...
				dwResult = GetLastError();
				if(dwResult == ERROR_NONE_MAPPED)
					wcscpy_s(lpName, sizeof(lpName),
L"NONE_MAPPED");
				else
				{
					wprintf(L"LookupAccountSid() failed, error
%u\n", GetLastError());
					return FALSE;
				}
			}
			// If found...
			else
			{
				//****************** Built-in\Administrators group
********************
				wprintf(L"BUILTIN\\Administrators group SID
found!\n");
				wprintf(L"YES, current user is a member of the %s\\%s
group\n", lpDomain, lpName);

				//****************** Local group
************************************
				// wprintf(L"LookupAccountSid() for Local group is
OK\n");
				// wprintf(L"Current user is a member of the %s
group\n", lpName);
				//****************** Authenticated users
*****************************
				// wprintf(L"LookupAccountSid() for Authenticated
users is OK\n");
```

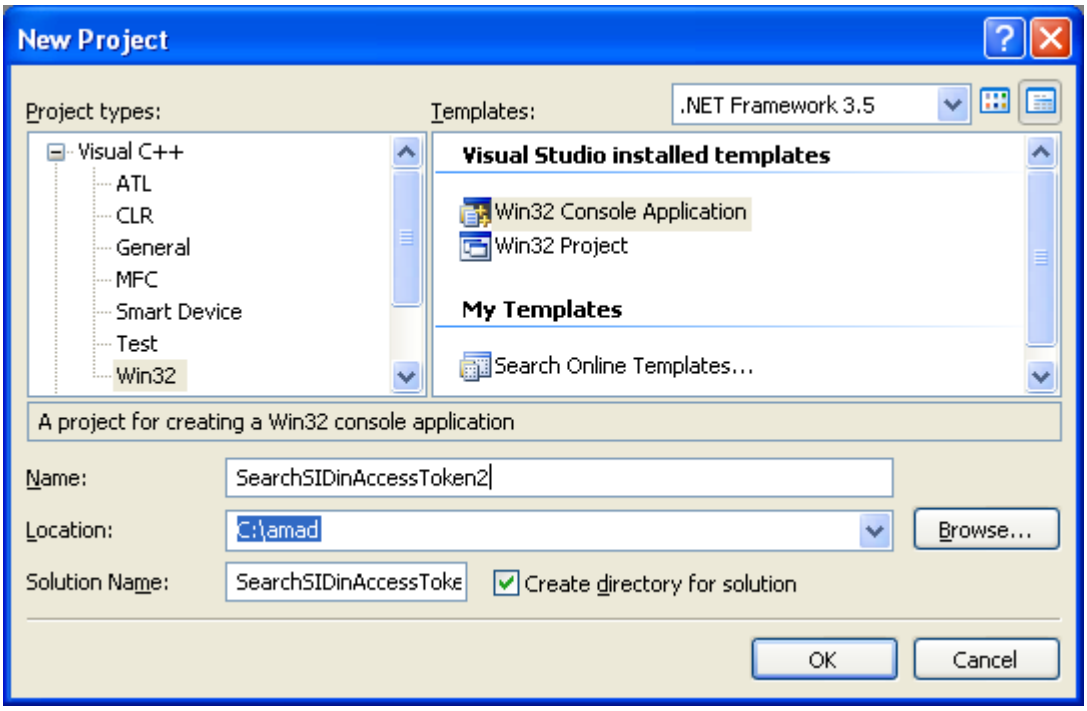
```
                        // wprintf(L"Current user is a member of the %s\\%s
group\n", lpDomain, lpName);
                  }
                  //*************** End playing with SIDs
*******************************

                  // Find out whether the SID is enabled in the token.
                  if(pGroupInfo->Groups[i].Attributes & SE_GROUP_ENABLED)
                        wprintf(L" and the group SID is enabled!\n");
                  else if (pGroupInfo->Groups[i].Attributes &
SE_GROUP_USE_FOR_DENY_ONLY)
                        wprintf(L" and the group SID is a deny-only SID!\n");
                  else
                        wprintf(L"and the group SID is not enabled!\n");
            }
      }

      // Release resources back to system
      if(pSID)
            FreeSid(pSID);
      if(pGroupInfo)
            GlobalFree(pGroupInfo);

      return TRUE;
}

//******** main() ********
int wmain(int argc, WCHAR **argv)
{
      // Call the user defined SearchTokenGroupsForSID() function to search
the token group SID
      BOOL bRetVal = SearchTokenGroupsForSID();

      // Verify
      wprintf(L"The return value of SearchTokenGroupsForSID() is: %d\n",
bRetVal);

      return 0;
}
```

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

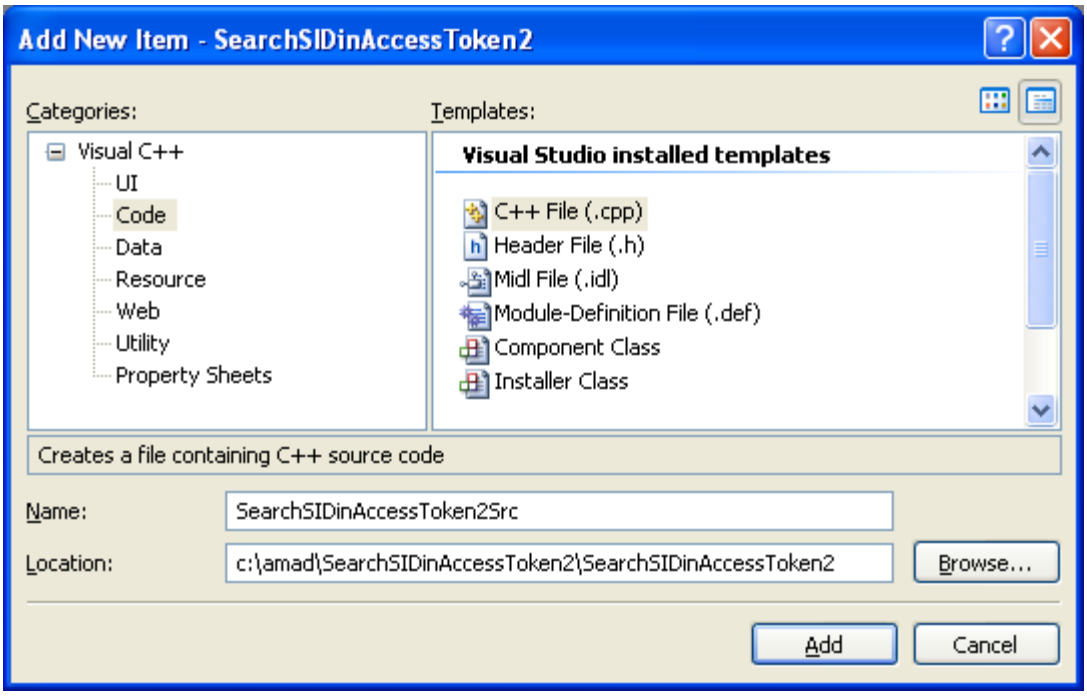


Searching for a SID in an Access Token Program Example 2

The following program example is another working sample which is a smaller than the previous one. Notice the functions used in this program and compared to the previous program example. 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.

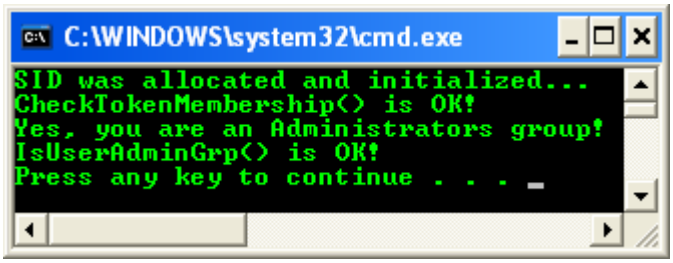


Next, add the following source code.

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

/*
This routine returns TRUE if the caller's process is a member of the
Administrators local group. Caller is NOT expected
to be impersonating anyone and is expected to be able
to open its own process and process token.

Arguments: None.
Return Value:
   TRUE - Caller has Administrators local group.
   FALSE - Caller does not have Administrators local group.
*/
BOOL IsUserAdminGrp(void)
{
      BOOL check = FALSE;
      SID_IDENTIFIER_AUTHORITY NtAuthority = SECURITY_NT_AUTHORITY;
      PSID AdministratorsGroup;

      check = AllocateAndInitializeSid(
            &NtAuthority,
            2,
            SECURITY_BUILTIN_DOMAIN_RID,
            DOMAIN_ALIAS_RID_ADMINS,
            0, 0, 0, 0, 0, 0,
            &AdministratorsGroup);

      // if TRUE
      if(check)
```

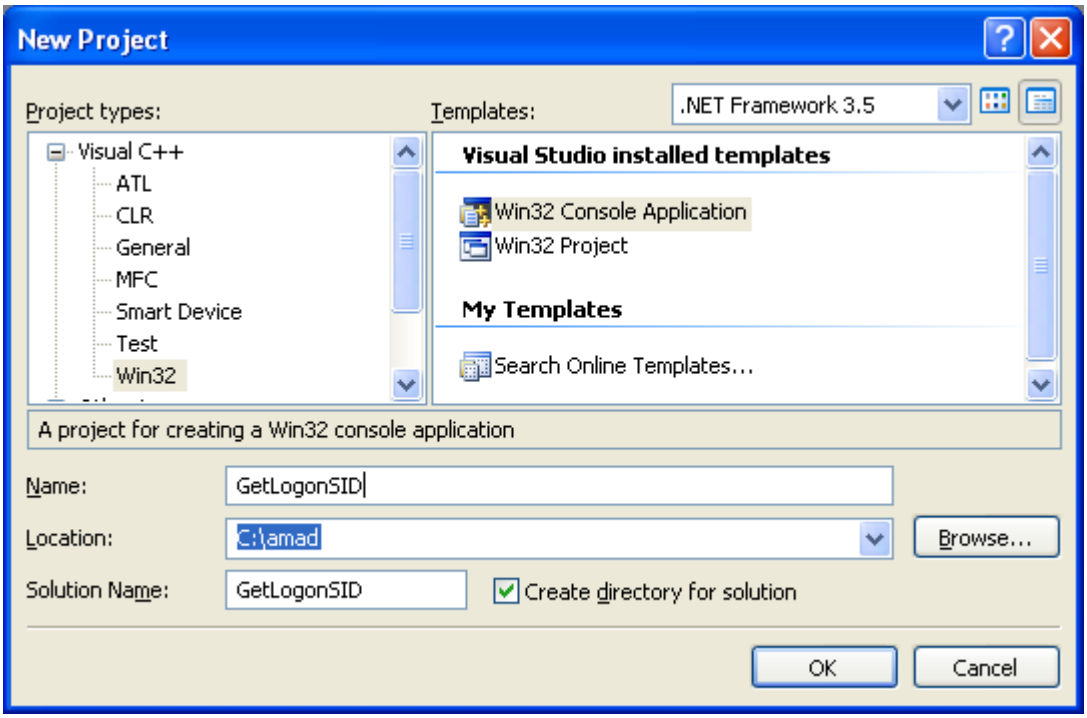
```
      {
            wprintf(L"SID was allocated and initialized...\n");
            // Determines whether a specified security identifier (SID) is
enabled in an access token.
            if(!CheckTokenMembership(
                  NULL, // uses the impersonation token of the calling
thread.
                              // If the thread is not impersonating, the
function duplicates
                              // the thread's primary token to create an
impersonation token
                  AdministratorsGroup,    // Pointer to a SID structure
                  &check                        // Result of the SID
                  ))
            {
                  wprintf(L"CheckTokenMembership() failed, error %u\n",
GetLastError());
            }
            else
            {
                  wprintf(L"CheckTokenMembership() is OK!\n");
                  // If the SID (the 2nd parameter) is present and has the
SE_GROUP_ENABLED attribute,
                  // check (3rd parameter) returns TRUE; otherwise, it
returns FALSE.
                  if(check == TRUE)
                        wprintf(L"Yes, you are an Administrators group!\n");
                  else
                        wprintf(L"No, you are not an Administrators
group!\n");
            }
            FreeSid(AdministratorsGroup);
      }
      else
            wprintf(L"AllocateAndInitializeSid() failed, error %u\n",
GetLastError());

      return(check);
}

int wmain(int argc, WCHAR **argv)
{
       BOOL bRetVal = IsUserAdminGrp();

         if(!bRetVal)
               wprintf(L"IsUserAdminGrp() failed, error %u\n",
GetLastError());
         else
               wprintf(L"IsUserAdminGrp() is OK!\n");

       return 0;
}
```

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

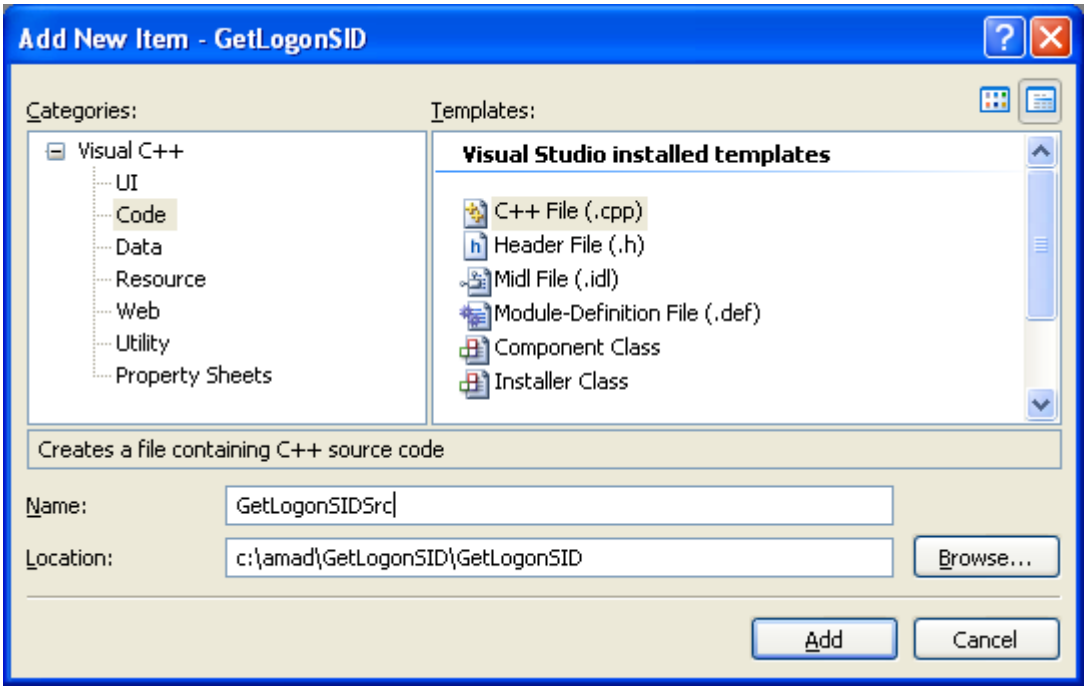


Getting the Logon (Session) SID in C++

A logon security identifier (SID) identifies the logon session associated with an access token. A typical use of a logon SID is in an ACE that allows access for the duration of a client's logon session. For example, a Windows service can use the LogonUser() function to start a new logon session. The LogonUser function returns an access token from which the service can extract the logon SID. The service can then use the SID in an ACE that allows the client's logon session to access the interactive window station and desktop.
The following program example gets the logon SID from an access token. It uses the GetTokenInformation() function to fill a TOKEN_GROUPS buffer with an array of the group SIDs from an access token. This array includes the logon SID, which is identified by the SE_GROUP_LOGON_ID attribute. The example function allocates a buffer for the logon SID; it is the caller's responsibility to free the buffer.
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.

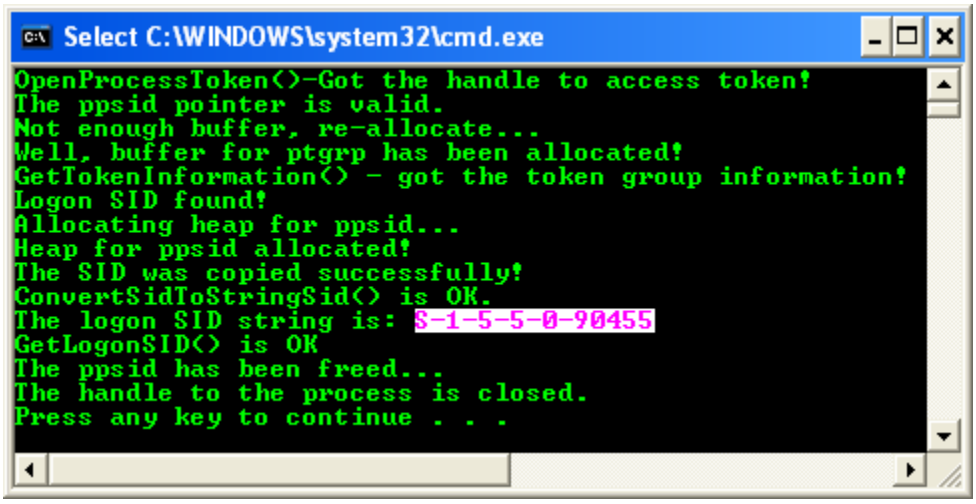
```
// Getting the current Logon SID
#include <windows.h>
#include <stdio.h>
// For SID conversion
#include <sddl.h>
#include <aclapi.h>

// Simple clean up routine
void Cleanup(PTOKEN_GROUPS ptgrp)
{
	// Release the buffer for the token groups.
	if(ptgrp != NULL)
	{
		wprintf(L"Freeing up the ptgrp buffer...\n");
		HeapFree(GetProcessHeap(), 0, (LPVOID)ptgrp);
	}
	else
		wprintf(L"ptgrp buffer has been freed-up!\n");
}

// Get the logon SID and convert it to SID string...
BOOL GetLogonSID(HANDLE hToken, PSID ppsid)
{
	BOOL bSuccess = FALSE;
	DWORD dwIndex;
	DWORD dwLength = 0;
```

```
	PTOKEN_GROUPS ptgrp = NULL;
	// Dummy initialization...
	LPTSTR pSid = L"";

	// Verify the parameter passed in is not NULL.
	// Although we just provide an empty buffer...
	if(ppsid == NULL)
	{
		wprintf(L"The ppsid pointer is NULL lol!\n");
		Cleanup(ptgrp);
	}
	else
		wprintf(L"The ppsid pointer is valid.\n");

	// Get the 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) ptgrp,	// pointer to TOKEN_GROUPS buffer
		0,				// size of buffer
		&dwLength		// receives required buffer size
		))
	{
		if(GetLastError() != ERROR_INSUFFICIENT_BUFFER)
		{
			wprintf(L"GetTokenInformation() - buffer is OK!\n");
			Cleanup(ptgrp);
		}
		else
		{
			wprintf(L"Not enough buffer, re-allocate...\n");
			ptgrp = (PTOKEN_GROUPS)HeapAlloc(GetProcessHeap(),
HEAP_ZERO_MEMORY, dwLength);
		}

		if(ptgrp == NULL)
		{
			wprintf(L"Failed to allocate heap for ptgrp, error %u\n",
GetLastError());
			Cleanup(ptgrp);
		}
		else
			wprintf(L"Well, buffer for ptgrp has been allocated!\n");
	}
	else
		wprintf(L"GetTokenInformation() is pretty fine!\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) ptgrp,	// pointer to TOKEN_GROUPS buffer
		dwLength,		// size of buffer
		&dwLength		// receives required buffer size
		))
	{
```

```
            wprintf(L"GetTokenInformation()  failed, error %u\n",
GetLastError());
            Cleanup(ptgrp);
      }
      else
            wprintf(L"GetTokenInformation() - got the token group
information!\n");

      // Loop through the groups to find the logon SID.
      for(dwIndex = 0; dwIndex < ptgrp->GroupCount; dwIndex++)
            if((ptgrp->Groups[dwIndex].Attributes & SE_GROUP_LOGON_ID) ==
SE_GROUP_LOGON_ID)
            {
                  wprintf(L"Logon SID found!\n");
                  // If the logon SID is found then make a copy of it.
                  dwLength = GetLengthSid(ptgrp->Groups[dwIndex].Sid);
                  // Allocate a storage
                  wprintf(L"Allocating heap for ppsid...\n");
                  ppsid = (PSID) HeapAlloc(GetProcessHeap(),
HEAP_ZERO_MEMORY, dwLength);
                  // and verify again...
                  if(ppsid == NULL)
                        Cleanup(ptgrp);
                  else
                        wprintf(L"Heap for ppsid allocated!\n");

                  // If Copying the SID fails...
                  if(!CopySid(dwLength,
                        ppsid,    // Destination
                        ptgrp->Groups[dwIndex].Sid))  // Source
                  {
                        wprintf(L"Failed to copy the SID, error %u\n",
GetLastError());
                        HeapFree(GetProcessHeap(), 0, (LPVOID)ppsid);
                        Cleanup(ptgrp);
                  }
                  else
                        wprintf(L"The SID was copied successfully!\n");

                  // Convert the logon sid to SID string format
                  if(!(ConvertSidToStringSid(
                        ppsid,  // Pointer to the SID structure to be
converted
                        &pSid))) // Pointer to variable that receives the
null-terminated SID string
                  {
                        wprintf(L"ConvertSidToStringSid() failed, error
%u\n", GetLastError());
                        Cleanup(ptgrp);
                        exit(1);
                  }
                  else
                  {
                        wprintf(L"ConvertSidToStringSid() is OK.\n");
                        wprintf(L"The logon SID string is: %s\n", pSid);
                  }
                  // The search was found, so break out from the loop
```

```
                    break;
            }

            LocalFree(pSid);
            // If everything OK, returns a clean slate...
            bSuccess = TRUE;
            return bSuccess;
}

// The following function release the buffer allocated by the GetLogonSID()
function.
BOOL FreeLogonSID(PSID ppsid)
{
      HeapFree(GetProcessHeap(), 0, (LPVOID)ppsid);
      return TRUE;
}

int wmain(int argc, WCHAR **argv)
{
      // Handle to token
      HANDLE hToken;

      // A 'dummy' initial size of SID to avoid a NULL pointer
      BYTE sidBuffer[256];
      PSID ppsid = (PSID)&sidBuffer;

      // Open a handle to the access token for the calling process
      // that is the currently login access token
      if(!OpenProcessToken(GetCurrentProcess(), TOKEN_ALL_ACCESS, &hToken))
      {
            wprintf(L"OpenProcessToken()-Getting the handle to access token
failed, error %u\n", GetLastError());
      }
      else
            wprintf(L"OpenProcessToken()-Got the handle to access token!\n");

      // Call the GetLogonSID()
      if(GetLogonSID(hToken, ppsid))
            wprintf(L"GetLogonSID() is OK\n");
      else
            wprintf(L"GetLogonSID() failed, error %u\n\n", GetLastError());

      // Release the allocation for ppsid
      if(FreeLogonSID(ppsid))
            wprintf(L"The ppsid has been freed...\n");
      // Close the handle lol
      if(CloseHandle(hToken))
            wprintf(L"The handle to the process is closed.\n");
      return 0;
}
```

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



The SID string is in the S-1-5-5-X-Y format which indicates a logon session. The X and Y values for these SIDs are different for each session. The well-known SID string for user, group and domain can be found at Windows well-known SID.
The SID string is S-1-5-5-0-90455 is a logon session SID for the machine that used to run the program example. This is used for process in a given logon session, gaining access to the window-station objects for that session. The constant format is S-1-5-5-X-Y (SECURITY_LOGON_IDS_RID). The X and Y values for these SIDs are different for each logon session. The value of the SECURITY_LOGON_IDS_RID_COUNT is the number of RIDs in this identifier (5-X-Y).