Windows Access Control List (ACL) 3

What do we have in this session?

1. Security Identifiers
2. Interaction Between Threads and Securable Objects
3. DACLs and ACEs
4. How DACLs Control Access to an Object
5. Order of ACEs in a DACL
6. ACEs to Control Access to an Object's Properties
7. Requesting Access Rights to an Object
8. Null DACLs and Empty DACLs
9. Allowing Anonymous Access
10. Security Descriptor Definition Language
11. SDDL Examples
12. Security Descriptor String Examples

Abilities that should be acquired in this session are:

1. Able to understand SID related terms and their meanings.
2. Able to understand Windows Access Token, Securable Object and Security Descriptor.
3. Able to understand relate the Trustee and Security Identifier (SID).
4. Able to understand Discretionary Access Control List (DACL) & Security Access Control List (SACL).
5. Able to understand Access Control Entry (ACE) and their components.

Security Identifiers (SID)

A security identifier (SID) is **a unique value of variable length used to identify a trustee**. Each account has a unique SID issued by an authority, such as a Windows domain controller, and stored in a **security database**. Each time a user logs on, the system retrieves the SID for that user from the database and places it in the access token for that user. The system uses the SID in the access token to identify the user in all subsequent interactions with Windows security. When a SID has been used as the unique identifier for a user or group, it cannot ever be used again to identify another user or group. Windows security uses SIDs in the following security elements:

1. In security descriptors to identify the owner of an object and primary group.
2. In access control entries (ACEs), to identify the trustee for whom access is allowed, denied, or audited.

3. In access tokens, to identify the user and the groups to which the user belongs.

In addition to the uniquely created, domain-specific SIDs assigned to specific users and groups, there are **well-known SIDs that identify generic groups and generic users**. For example, the well-known SIDs, Everyone and World, identify a group that includes all users. Most applications never need to work with SIDs. For example, in Windows NT 4.0, the security functions for getting and setting the ACEs in an ACL allow you to use names rather than SIDs to identify users and groups. Because the names of well-known SIDs can vary, you should use the functions to build the SID from predefined constants rather than using the name of the well-known SID. For example, the U.S. English version of the Windows operating system has a well-known SID named "BUILTIN\Administrators" that might have a different name on international versions of the system. If you do need to work with SIDs, do not manipulate them directly. Instead, use the following functions.

Function	Description
AllocateAndInitializeSid()	Allocates and initializes a SID with the specified number of sub authorities.
ConvertSidToStringSid()	Converts a SID to a string format suitable for display, storage, or transport.
ConvertStringSidToSid()	Converts a string-format SID to a valid, functional SID.
CopySid()	Copies a source SID to a buffer.
EqualPrefixSid()	Tests two SID prefix values for equality. A SID prefix is the entire SID except for the last sub authority value.
EqualSid()	Tests two SIDs for equality. They must match exactly to be considered equal.
FreeSid()	Frees a previously allocated SID by using the AllocateAndInitializeSid() function.
GetLengthSid()	Retrieves the length of a SID.
GetSidIdentifierAuthority()	Retrieves a pointer to the identifier authority for a SID.
GetSidLengthRequired()	Retrieves the size of the buffer required to store a SID with a specified number of sub-authorities.
GetSidSubAuthority()	Retrieves a pointer to a specified sub authority in a SID.
GetSidSubAuthorityCount()	Retrieves the number of sub authorities in a SID.
InitializeSid()	Initializes a SID structure.
IsValidSid()	Tests the validity of a SID by verifying that the revision number is within a known range and that the number of sub authorities is less than the maximum.
LookupAccountName()	Retrieves the SID that corresponds to a specified account name.

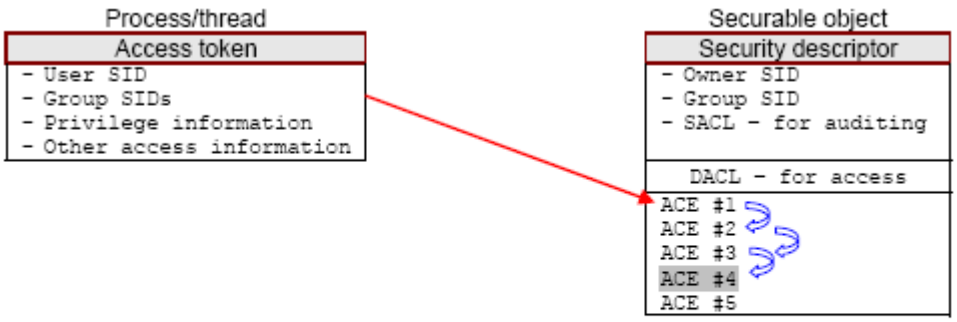
LookupAccountSid()	Retrieves the account name that corresponds to a specified SID.

Table 16

Interaction Between Threads and Securable Objects

An application consists of one or more processes. A process, in the simplest terms, is an executing program. One or more threads run in the context of the process. **A thread is the basic unit to which the operating system allocates processor time**. A thread can execute any part of the process code, including parts currently being executed by another thread. A fiber is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them. A job object allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object. When a thread attempts to use a securable object, the system performs an access check before allowing the thread to proceed. In an access check, the system compares the security information in the thread's access token against the security information in the object's security descriptor. The access token contains SIDs that identify the user associated with the thread.

The security descriptor identifies the object's owner and contains a DACL. The DACL contains ACEs, each of which specify the access rights allowed or denied to a specific user or group. The system checks the object's DACL, looking for ACEs that apply to the user and group SIDs from the thread's access token. The system checks each ACE until access is either granted or denied or until there are no more ACEs to check. Conceivably, an ACL could have several ACEs that apply to the token's SIDs and, if this occurs, the access rights granted by each ACE accumulate. For example, if one ACE grants read access to a group and another ACE grants write access to a user who is a member of the group, the user can have both read and write access to the object. The following illustration shows the relationship between these blocks of security information.



The system checks each ACE in the DACL until access is granted or denied, or until there are no more ACEs. If ACE #4 resolves the access-control, the system does not check the last ACE.

DACLs and ACEs

The situations encountered when checking the ACEs in DACLs are listed below:

1. If a Windows object **does not have a DACL**, the system allows everyone full access to it.
2. If an object has a DACL, the system allows only the access that is explicitly allowed by the ACEs in the DACL.
3. If there are **no ACEs in the DACL**, the system does not allow access to anyone.
4. If a DACL has ACEs that allow access to a limited set of users or groups, the system implicitly denies access to all trustees not included in the ACEs.

In most cases, you can control access to an object by using **access-allowed ACEs**; you do not need to explicitly deny access to an object. The exception is when an ACE allows access to a group and you want to deny access to a member of the group. To do this, place an access-denied ACE for the user in the DACL ahead of the access-allowed ACE for the group. Note that the order of the ACEs is important because the system reads the ACEs in sequence until access is granted or denied. The user's access-denied ACE must appear first; otherwise, when the system reads the group's access allowed ACE, it will grant access to the restricted user.

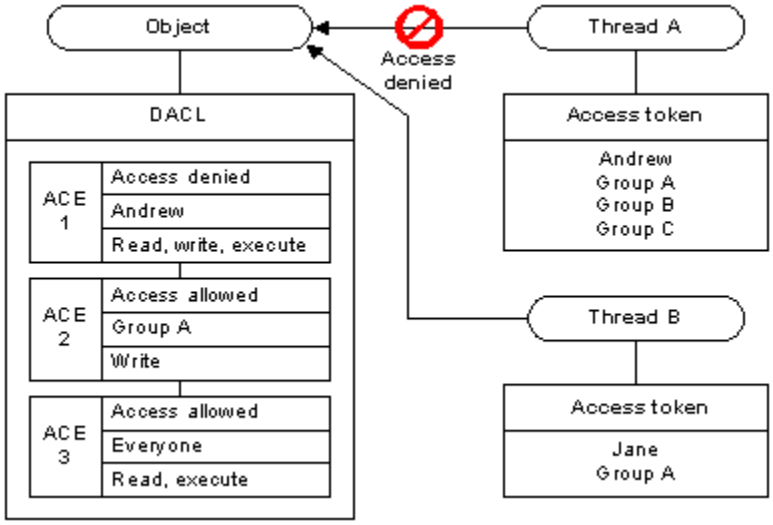
How DACLs Control Access to an Object

When a thread tries to access a securable object, the system either grants or denies access. If the object does not have a DACL, the system grants access; otherwise, the system looks for ACEs in the object's DACL that apply to the thread. Each ACE in the object's DACL specifies the access rights allowed or denied for a trustee, which can be a user account, a group account, or a logon session. The system compares the trustee in each ACE to the trustees identified in the thread's

access token. An access token contains SIDs that identify the user and the group accounts to which the user belongs. A token also contains a logon SID that identifies the current logon session. During an access check, the system ignores group SIDs that are not enabled. Typically, the system uses the primary access token of the thread that is requesting access. However, if the thread is impersonating another user, the system uses the thread's impersonation token. The system examines each ACE in sequence until one of the following events occurs:

1. An access-denied ACE explicitly denies any of the requested access rights to one of the trustees listed in the thread's access token.
2. One or more access-allowed ACEs for trustees listed in the thread's access token explicitly grant all the requested access rights.
3. All ACEs have been checked and there is still at least one requested access right that has not been explicitly allowed, in which case, access is implicitly denied.

The following illustration shows how an object's DACL can allow access to one thread while denying access to another.



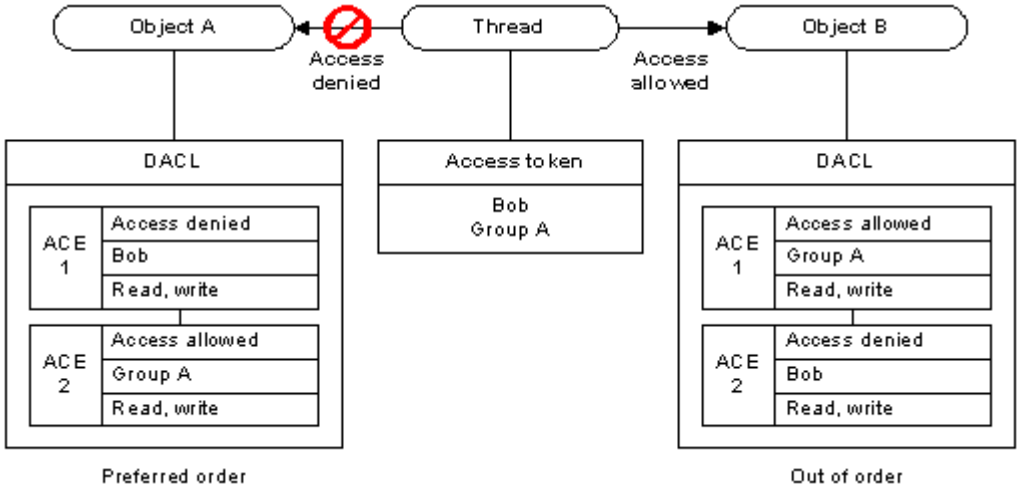
For Thread A, the system reads ACE 1 and immediately denies access because the access-denied ACE applies to the user in the thread's access token. In this case, the system does not check ACEs 2 and 3. For Thread B, ACE 1 does not apply, so the system proceeds to ACE 2, which allows write access, and ACE 3 which allows read and execute access. Because the system stops checking ACEs when the requested access is explicitly granted or denied, the order of ACEs in a DACL is important. Note that if the ACE order were different in the example, the system might have granted access to Thread A. For system objects, the operating system defines a preferred order of ACEs in a DACL.

Order of ACEs in a DACL

When a process tries to access a securable object, the system steps through the ACEs in the object's DACL until it finds ACEs that allow or deny the requested access. The access rights that a DACL allows a user could vary depending on the order of ACEs in the DACL. Consequently, the Windows XP/Windows 2000/Windows NT operating systems define a preferred order for ACEs in the DACL of a securable object. The preferred order provides a simple framework that ensures that an access-denied ACE actually denies access. On Windows Server 2003, Windows XP, and Windows 2000, the proper order of ACEs is complicated by the introduction of object-specific ACEs and automatic inheritance. The following steps describe the preferred order:

1. All explicit ACEs are placed in a group before any inherited ACEs.
2. Within the group of explicit ACEs, access-denied ACEs are placed before access-allowed ACEs.
3. Inherited ACEs are placed in the order in which they are inherited. ACEs inherited from the child object's parent come first, and then ACEs inherited from the grandparent, and so on up the tree of objects.
4. For each level of inherited ACEs, access-denied ACEs are placed before access-allowed ACEs.

Take note that not all ACE types are required in an ACL. Functions such as AddAccessAllowedAceEx() and AddAccessAllowedObjectAce() add an ACE to the end of an ACL. It is the caller's responsibility to ensure that the ACEs are added in the proper order. On Windows NT 4.0 and earlier, the preferred order of ACEs is simple: In a DACL, all access-denied ACEs should precede any access-allowed ACEs. The SetEntriesInAcl() function creates a DACL with ACEs in this order. However, the low-level functions for adding ACEs to a DACL do not enforce the preferred order. The AddAce() function adds ACEs at a specified location in an ACL. Functions such as AddAccessAllowedAce() add an ACE to the end of an ACL. It is the caller's responsibility to ensure that the ACEs are added in the preferred order. The following illustration shows how the same ACEs can allow different access rights, depending on their order in the DACL. The system denies access to Object A when it reads the access-denied ACE, but the out-of-order DACL for Object B causes the system to grant access without reading the access-denied ACE.

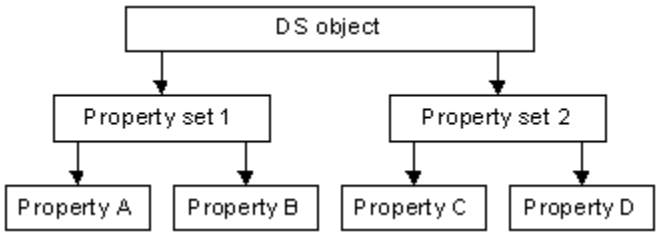


ACEs to Control Access to an Object's Properties

The DACL of a directory service (DS) object can contain a hierarchy of ACEs, as follows:

1. ACEs that protect the object itself.
2. Object-specific ACEs that protect a specified property set on the object.
3. Object-specific ACEs that protect a specified property on the object.

Within this hierarchy, the rights granted or denied at a higher level apply also to the lower levels. For example, if an object-specific ACE on a property set allows a trustee the ADS_RIGHT_DS_READ_PROP right, the trustee has implicit read access to all of the properties of that property set. Similarly, an ACE on the object itself that allows ADS_RIGHT_DS_READ_PROP access gives the trustee read access to all of the object's properties. The following illustration shows the tree of a hypothetical DS object and its property sets and properties.



Suppose you want to allow the following access to the properties of this DS object:

1. Allow Group A read/write access to all of the object's properties.
2. Allow everyone else read/write access to all properties except Property D.

To do this, set the ACEs in the object's DACL as shown in the following table.

Trustee	Object GUID	ACE type	Access rights
Group A	None	Access-allowed ACE	ADS_RIGHT_DS_READ_PROP | ADS_RIGHT_DS_WRITE_PROP
Everyone	Property Set 1	Access-allowed object ACE	ADS_RIGHT_DS_READ_PROP | ADS_RIGHT_DS_WRITE_PROP
Everyone	Property C	Access-allowed object ACE	ADS_RIGHT_DS_READ_PROP | ADS_RIGHT_DS_WRITE_PROP

Table 17

The ACE for Group A does not have an object GUID, which means that it allows access to all the object's properties. The object-specific ACE for Property Set 1 allows everyone access to Properties A and B. The other object-specific ACE allows everyone access to Property C. Note that although this DACL does not have any access-denied ACEs, it implicitly denies Property D access to everyone except Group A. When a user tries to access an object's property, the system checks the ACEs, in order, until the requested access is explicitly granted, denied, or there are no more ACEs, in which case, access is implicitly denied. The system evaluates:

1. ACEs that apply to the object itself.
2. Object-specific ACEs that apply to the property set that contains the property being accessed.
3. Object-specific ACEs that apply to the property being accessed.

The system ignores object-specific ACEs that apply to other property sets or properties.

Requesting Access Rights to an Object

When you open a handle to an object, the returned handle has some combination of access rights to the object. Some functions, such as CreateSemaphore(), do not require a specific set of requested access rights. These functions always try to open the handle for full access. Other functions, such as CreateFile() and OpenProcess(), allow you to specify the set of access rights that you want. You should request only the access rights that you need, rather than opening a handle for full access. This prevents using the handle in an unintended way, and increases the chances that the access request will succeed if the object's DACL only allows limited access. You should use generic access rights to specify the type of access needed when opening a handle to an object. This is typically simpler than specifying all the corresponding standard and specific

rights. Alternatively, use the MAXIMUM_ALLOWED constant to request that the object be opened with all the access rights that are valid for the caller. Take note that the MAXIMUM_ALLOWED constant cannot be used in an ACE. To get or set the SACL in an object's security descriptor, request the ACCESS_SYSTEM_SECURITY access right when opening a handle to the object.

Null DACLs and Empty DACLs

If the DACL belonging to an object's security descriptor is set to NULL, a null DACL is created. **A null DACL grants full access to any user that requests it that is full access for Everyone and normal security checking is not performed with respect to the object** (newer Win32 API has fixed this issue). A null DACL should not be confused with an empty DACL. An empty DACL is a properly allocated and initialized DACL containing no ACEs. **An empty DACL grants no access to the object it is assigned to**.

Allowing Anonymous Access

The default security policy restricts anonymous local access to having no rights. Administrators can then add or subtract rights as they see fit. On Windows NT4 the Anonymous access allows access equal to the access granted to the Everyone group. A local access group exists for applications with the same access rights as Everyone (equivalent to Windows NT anonymous access). Administrators can then appropriately increase or decrease the number of users in that group, named the Pre-Windows 2000-Compatible Access Group.

Security Descriptor Definition Language (SDDL)

The security descriptor definition language (sddl) defines the string format that the ConvertSecurityDescriptorToStringSecurityDescriptor() and ConvertStringSecurityDescriptorToSecurityDescriptor() functions use to describe a security descriptor as a text string. The language also defines string elements for describing information in the components of a security descriptor.

Security Descriptor String Format

The Security Descriptor String Format is a text format for storing or transporting information in a security descriptor as shown in the following example.

```
"O:AOG:DAD:(A;;RPWPCCDCLCSWRCWDWOGA;;;S-1-0-0)
S:(AU;SAFA;WDWOSDWPCCDCSW;;;WD)"
```

The ConvertSecurityDescriptorToStringSecurityDescriptor() and ConvertStringSecurityDescriptorToSecurityDescriptor() functions use this format. The format is a null-terminated string with tokens to indicate each of the four main components of a security descriptor:

1. Owner (O:).
2. Primary group (G:).
3. DACL (D:), and
4. SACL (S:).

The format:

1. O:owner_sid
2. G:group_sid
3. D:dacl_flags(string_ace1)(string_ace2)... (string_aceN)
4. S:sacl_flags(string_ace1)(string_ace2)... (string_aceN)

The SID description:

1. owner_sid - A SID string that identifies the object's owner.
2. group_sid - A SID string that identifies the object's primary group.
3. dacl_flags - Security descriptor control flags that apply to the DACL. The dacl_flags string can be a concatenation of zero or more of the following strings:

Control	Constant in sddl.h	Meaning
"P"	SDDL_PROTECTED	The SE_DACL_PROTECTED flag is set.
"AR"	SDDL_AUTO_INHERIT_REQ	The SE_DACL_AUTO_INHERIT_REQ flag is set.
"AI"	SDDL_AUTO_INHERITED	The SE_DACL_AUTO_INHERITED flag is set.

Table 18

4. sacl_flags - Security descriptor control flags that apply to the SACL. The sacl_flags string uses the same control bit strings as the dacl_flags string.
5. string_ace - A string that describes an ACE in the security descriptor's DACL or SACL. Each ACE string is enclosed in parentheses, ( ).

Unneeded components can be omitted from the security descriptor string. For example, if the SE_DACL_PRESENT flag is not set in the input security descriptor, ConvertSecurityDescriptorToStringSecurityDescriptor() does not include a D: component in the output string. You can also use the SECURITY_INFORMATION bit flags to indicate the

components to include in a security descriptor string. The security descriptor string format does not support NULL ACLs. **To denote an empty ACL, the security descriptor string includes the D: or S: token with no additional string information**. The security descriptor string stores the SECURITY DESCRIPTOR CONTROL bits in different ways. The SE_DACL_PRESENT or SE_SACL_PRESENT bits are indicated by the presence of the D: or S: token in the string. Other bits that apply to the DACL or SACL are stored in dacl_flags and sacl_flags. The SE_OWNER_DEFAULTED, SE_GROUP_DEFAULTED, SE_DACL_DEFAULTED, and SE_SACL_DEFAULTED bits are not stored in a security descriptor string. The SE_SELF_RELATIVE bit is not stored in the string, but ConvertStringSecurityDescriptorToSecurityDescriptor() always sets this bit in the output security descriptor.

Security Descriptor String Examples

The following examples show security descriptor strings and the information in the associated security descriptors.

String 1 example:

```
"O:AOG:DAD:(A;;RPWPCCDCLCSWRCWDWOGA;;;S-1-0-0)"

Security Descriptor 1:
   Revision:  0x00000001
   Control:   0x0004
      SE_DACL_PRESENT
   Owner: (S-1-5-32-548)
   PrimaryGroup: (S-1-5-21-397955417-626881126-188441444-512)
DACL
   Revision: 0x02
   Size:     0x001c
   AceCount: 0x0001
   Ace[00]
      AceType:      0x00 (ACCESS_ALLOWED_ACE_TYPE)
      AceSize:      0x0014
      InheritFlags:  0x00
      Access Mask:   0x100e003f
                     READ_CONTROL
                     WRITE_DAC
                     WRITE_OWNER
                     GENERIC_ALL
```

```
                    Others(0x0000003f)
      Ace Sid      : (S-1-0-0)
SACL
   Not present
```

String 2 example:

```
"O:DAG:DAD:(A;;RPWPCCDCLCRCWOWDSDSW;;;SY)(A;;RPWPCCDCLCRCWOWDSDSW;;;DA)(OA;;CCDC;bf967aba-0de6-11d0-a285-00aa003049e2;;AO)(OA;;CCDC;bf967a9c-0de6-11d0-a285-00aa003049e2;;AO)(OA;;CCDC;6da8a4ff-0e52-11d0-a286-00aa003049e2;;AO)(OA;;CCDC;bf967aa8-0de6-11d0-a285-00aa003049e2;;PO)(A;;RPLCRC;;;AU)S:(AU;SAFA;WDWOSDWPCCDCSW;;;WD)"

Security Descriptor 2:

   Revision:  0x00000001
   Control:   0x0014
      SE_DACL_PRESENT
      SE_SACL_PRESENT
   Owner: (S-1-5-21-397955417-626881126-188441444-512)
   PrimaryGroup: (S-1-5-21-397955417-626881126-188441444-512)
DACL
   Revision: 0x04
   Size:     0x0104
   AceCount: 0x0007
   Ace[00]
      AceType:      0x00 (ACCESS_ALLOWED_ACE_TYPE)
      AceSize:      0x0014
      InheritFlags:  0x00
      Access Mask:   0x000f003f
                     DELETE
                     READ_CONTROL
                     WRITE_DAC
                     WRITE_OWNER
                     Others(0x0000003f)
      Ace Sid:      (S-1-5-18)
   Ace[01]
      AceType:      0x00 (ACCESS_ALLOWED_ACE_TYPE)
      AceSize:      0x0024
```

```
   InheritFlags:  0x00
   Access Mask:   0x000f003f
                  DELETE
                  READ_CONTROL
                  WRITE_DAC
                  WRITE_OWNER
                  Others(0x0000003f)
   Ace Sid:       (S-1-5-21-397955417-626881126-188441444-512)
Ace[02]
   AceType:       0x05 (ACCESS_ALLOWED_OBJECT_ACE_TYPE)
   AceSize:       0x002c
   InheritFlags:  0x00
   Access Mask:   0x00000003
                  Others(0x00000003)
   Flags:         0x00000001, ACE_OBJECT_TYPE_PRESENT
   ObjectType:    GUID_C_USER
   InhObjectType: GUID ptr is NULL
   Ace Sid:       (S-1-5-32-548)
Ace[03]
   AceType:       0x05 (ACCESS_ALLOWED_OBJECT_ACE_TYPE)
   AceSize:       0x002c
   InheritFlags:  0x00
   Access Mask:   0x00000003
                  Others(0x00000003)
   Flags:         0x00000001, ACE_OBJECT_TYPE_PRESENT
   ObjectType:    GUID_C_GROUP
   InhObjectType: GUID ptr is NULL
   Ace Sid:       (S-1-5-32-548)
Ace[04]
   AceType:       0x05 (ACCESS_ALLOWED_OBJECT_ACE_TYPE)
   AceSize:       0x002c
   InheritFlags:  0x00
   Access Mask:   0x00000003
                  Others(0x00000003)
   Flags:         0x00000001, ACE_OBJECT_TYPE_PRESENT
   ObjectType:    GUID_C_LOCALGROUP
   InhObjectType: GUID ptr is NULL
   Ace Sid:       (S-1-5-32-548)
Ace[05]
   AceType:       0x05 (ACCESS_ALLOWED_OBJECT_ACE_TYPE)
```

AceSize: 0x002c
InheritFlags: 0x00
Access Mask: 0x00000003
Others(0x00000003)
Flags: 0x00000001, ACE_OBJECT_TYPE_PRESENT
ObjectType: GUID_C_PRINT_QUEUE
InhObjectType: GUID ptr is NULL
Ace Sid: (S-1-5-32-550)
Ace[06]
AceType: 0x00 (ACCESS_ALLOWED_ACE_TYPE)
AceSize: 0x0014
InheritFlags: 0x00
Access Mask: 0x00020014
READ_CONTROL
Others(0x00000014)
Ace Sid: (S-1-5-11)
SACL
Revision: 0x02
Size: 0x001c
AceCount: 0x0001
Ace[00]
AceType: 0x02 (SYSTEM_AUDIT_ACE_TYPE)
AceSize: 0x0014
InheritFlags: 0xc0
SUCCESSFUL_ACCESS_ACE_FLAG
FAILED_ACCESS_ACE_FLAG
Access Mask: 0x000d002b
DELETE
WRITE_DAC
WRITE_OWNER
Others(0x0000002b)
Ace Sid: (S-1-1-0)