Windows Access Control List (ACL) 7

What do we have in this session?

Abilities that supposed to be acquired in this session are:

1. Able to understand terms SACL and audit generation.
2. Able to understand and manipulate Impersonation.
3. Able to enable the auditing and viewing logging.

Audit Generation

The Windows API provides functions enabling an administrator to monitor security-related events. The security descriptor for a securable object can have a SACL. A **SACL contains ACEs that specify the types of access attempts that generate audit reports**. Each ACE identifies a trustee, a set of access rights, and a set of flags that indicate whether the system generates audit messages for **failed access attempts, successful access attempts, or both**. The system writes audit messages to the **security event log**. To read or write an object's SACL, a thread must first enable the SE_SECURITY_NAME privilege. The Windows API also provides support for server applications to generate audit messages when a client tries to access a private object.

SACL Access Right

The ACCESS_SYSTEM_SECURITY access right controls the ability to get or set the SACL in an object's security descriptor. The system grants this access right only if the SE_SECURITY_NAME privilege is enabled in the access token of the requesting thread. To access an object's SACL:

1. Call the AdjustTokenPrivileges() function to enable the SE_SECURITY_NAME privilege.
2. Request the ACCESS_SYSTEM_SECURITY access right when you open a handle to the object.
3. Get or set the object's SACL by using a function such as GetSecurityInfo() or SetSecurityInfo().
4. Call AdjustTokenPrivileges() to disable the SE_SECURITY_NAME privilege.

To access a SACL using the GetNamedSecurityInfo() or SetNamedSecurityInfo() functions, enable the SE_SECURITY_NAME privilege. The function internally requests the access right. The ACCESS_SYSTEM_SECURITY access right is not valid in a DACL because DACLs do not control access to a SACL. However, you can use the ACCESS_SYSTEM_SECURITY access right in a SACL to audit attempts to use the access right.

Auditing Access To Private Objects

A protected server can use the following functions to generate audit reports in the security event log (Administrative Tools ⟶ Event Viewer).

Function	Description
AccessCheckAndAuditAlarm()	Same as the AccessCheck() function except that it can generate audit messages for failed or successful access attempts.
AccessCheckByTypeAndAuditAlarm()	Same as the AccessCheckByType() function except that it can generate audit messages for failed or successful access attempts.
AccessCheckByTypeResultListAndAuditAlarm()	Same as the AccessCheckByTypeResultList() function except that it can generate audit messages for failed or successful access attempts.
AccessCheckByTypeResultListAndAuditAlarmByHandle()	Same as the AccessCheckByTypeResultListAndAuditAlarm() function except that it allows the calling thread to perform the access check before impersonating

	the client.
ObjectCloseAuditAlarm()	Generates an audit message to indicate that a client tried to close a private object.
ObjectDeleteAuditAlarm()	Generates an audit message to indicate that a client tried to delete a private object.
ObjectOpenAuditAlarm()	Generates an audit message to indicate that a client tried to open or create a private object.
ObjectPrivilegeAuditAlarm()	Generates an audit message to indicate that a client tried to use a specified set of privileges in conjunction with an attempt to access a private object.
PrivilegedServiceAuditAlarm()	Generates an audit message to indicate that a client tried to use a specified set of privileges.

Table 16

Low-level Access Control

Low-level security functions help you work with security descriptors, ACLs, ACEs and other related access control components.

Low-level Security Descriptor Functions

There are several pairs of low-level functions for setting and retrieving an object's security descriptor. Each of these pairs works only with a limited set of Windows objects. For example, one pair works with file objects and another works with registry keys. The following table shows the low-level functions to use with the different types of securable objects.

Object type	Low-level functions
Files, Directories, Mailslots, Named pipes.	Use the GetFileSecurity() and SetFileSecurity() functions. These functions use character strings to identify the securable object, instead of using handles.
Processes, Threads, Access tokens, File-mapping objects, Semaphores, Events, Mutexes, Waitable timers.	Use the GetKernelObjectSecurity() and SetKernelObjectSecurity() functions.

Window stations, Desktops.	Use the GetUserObjectSecurity() and SetUserObjectSecurity() functions.
Registry keys.	Use the RegGetKeySecurity() and RegSetKeySecurity() functions.
Windows service objects.	Use the QueryServiceObjectSecurity() and SetServiceObjectSecurity() functions.
Printer objects.	Use the PRINTER_INFO_2 structure with the GetPrinter() and SetPrinter() functions.
Network shares.	Use level 502 with the NetShareGetInfo() and NetShareSetInfo() functions.
Private objects (objects private to the creating application).	Use the CreatePrivateObjectSecurity(), DestroyPrivateObjectSecurity(), GetPrivateObjectSecurity() and SetPrivateObjectSecurity() functions.

Table 17

Low-level Security Descriptor Creation

Low-level access control **provides a set of functions for creating a security descriptor and getting and setting the components of a security descriptor**. The low-level functions for initializing and setting the components of a security descriptor work only with absolute-format security descriptors. The low-level functions for getting the components of a security descriptor work with both absolute and self-relative security descriptors. The InitializeSecurityDescriptor() function initializes a SECURITY_DESCRIPTOR buffer. The initialized security descriptor is in absolute format and has no owner, primary group, DACL, or SACL. You can use the following low-level functions to get or set specific components of a specified security descriptor.

Function	Description
GetSecurityDescriptorControl()	Retrieves revision and control information from a security descriptor.
GetSecurityDescriptorDacl()	Retrieves the DACL from a security descriptor.
GetSecurityDescriptorGroup()	Retrieves the primary group security identifier (SID) from a security descriptor.
GetSecurityDescriptorLength()	Returns the length of a security descriptor.
GetSecurityDescriptorOwner()	Retrieves the owner SID from a security descriptor.
GetSecurityDescriptorSacl()	Retrieves the SACL from a security descriptor.
SetSecurityDescriptorDacl()	Puts a DACL into a security descriptor, superseding any existing DACL.
SetSecurityDescriptorGroup()	Sets the primary group SID of a security descriptor.
SetSecurityDescriptorOwner()	Sets the owner SID of a security descriptor.

SetSecurityDescriptorSacl()	Puts a SACL into a security descriptor, superseding any existing SACL.

Table 18

To check the revision level and structural integrity of a security descriptor, call the IsValidSecurityDescriptor() function.

Absolute and Self-Relative Security Descriptors

A security descriptor can be in either absolute or self-relative format. In absolute format, a security descriptor contains pointers to its information, not the information itself. In self-relative format, a security descriptor stores a SECURITY_DESCRIPTOR structure and associated security information in a contiguous block of memory. To determine whether a security descriptor is self-relative or absolute, call the GetSecurityDescriptorControl() function and check the SE_SELF_RELATIVE flag of the SECURITY_DESCRIPTOR_CONTROL parameter. You can use the MakeSelfRelativeSD() and MakeAbsoluteSD() functions for converting between these two formats.
The absolute format is useful when you are building a security descriptor and have pointers to all of the components, for example, when default settings for the owner, group, and discretionary ACL are available. In this case, you can call the InitializeSecurityDescriptor() function to initialize a SECURITY_DESCRIPTOR structure, and then call functions such as SetSecurityDescriptorDacl() to assign ACL and SID pointers to the security descriptor. In self-relative format, a security descriptor always begins with a SECURITY_DESCRIPTOR structure, but the other components of the security descriptor can follow the structure in any order. Instead of using memory addresses, the security descriptor's components are identified by offsets from the beginning of the descriptor. This format is useful when a security descriptor must be stored on disk, transmitted by means of a communications protocol, or copied in memory. Except for MakeAbsoluteSD(), all functions that return a security descriptor do so using the self-relative format. Security descriptors passed as arguments to a function can be either self-relative or absolute form.

Low-level ACL and ACE Functions

To create an ACL using the low-level functions, allocate a buffer for the ACL and then initialize it by calling the InitializeAcl() function. To add ACEs to the end of a DACL, use the AddAccessAllowedAce() and AddAccessDeniedAce() functions. The AddAuditAccessAce() function adds an ACE to the end of a SACL. You can use the AddAce() function to add one or more ACEs at a specified position in an ACL. The AddAce() function also allows you to add an inheritable ACE to an ACL. The DeleteAce() function removes an ACE from a specified position in an ACL. The GetAce() function retrieves an ACE from a specified position in an ACL. The FindFirstFreeAce() function retrieves a pointer to the first free byte in an ACL. To modify an existing ACL in an object's security descriptor, use the GetSecurityDescriptorDacl() or GetSecurityDescriptorSacl() function to get the existing ACL. You can use the GetAce() function to copy ACEs from the existing ACL. After allocating and initializing a new ACL, use functions such as AddAccessAllowedAce() and AddAce() to add ACEs to it. When you have finished building the new ACL, use the SetSecurityDescriptorDacl() or SetSecurityDescriptorSacl() function to add the new ACL to the object's security descriptor. You can use the AddAccessAllowedObjectAce(), AddAccessDeniedObjectAce(), or AddAuditAccessObjectAce() functions to add object-specific ACEs to the end of an ACL.

How Security Groups are Used in Access Control

The security identifier is the **object identifier of the user or security group when the user or group is used for security purposes**. The name of the user or group is not used as the unique identifier within the system. The SID is stored in the objectSid attribute of user objects and security group objects. Active Directory generates the objectSid when the user or group is created. The system ensures that the SIDs are unique across a forest. Be aware that the objectGuid is the unique identifier of a user, group, or any other directory object. The SID changes if a user or group is moved to another domain; the objectGuid remains the same. When a user or group is given permission to access a resource, such as a printer or a file share, the SID of the user or group is added to the access control entry (ACE) defining the granted permission in the resource's discretionary access control list (DACL). In Active Directory, each object has an nTSecurityDescriptor attribute that stores a DACL defining the access to that particular object or attributes on that object. When a user logs on to a Windows 2000 domain, the operating system generates an access token. This access token is used to determine which resources the user may access. The user access token includes the following data:

1. User SID.
2. SIDs of all global and universal security groups that the user is a member of.
3. SIDs of all nested global and universal security groups.

Every process executed on behalf of this user has a copy of this access token. When the user attempts to access resources on a computer, the service through which the user accesses the resource will impersonate the user by creating a new access token based on the access token created at user logon time. This new access token will also contain the following SIDs:

1. SIDs for all domain local groups in the target domain that the user is a member of.
2. SIDs for all machine local groups on the target computer that the user is a member of.

The service uses this new access token to evaluate access to the resource. If a SID in the access token appears in any ACEs in the DACL, the service gives the user the permissions specified in those ACEs.

Impersonation

Impersonation is the **ability of a thread to execute in a security context that is different from the context of the process that owns the thread**. When running in the client's security context, the server "is" the client, to some degree. The server thread uses an access token representing the client's credentials to obtain access to the objects to which the client has access. The primary reason for impersonation is to cause access checks to be performed against the client's identity. Using the client's identity for access checks can cause access to be either restricted or expanded, depending on what the client has permission to do. For example, suppose a file server has files containing confidential information and that each of these files is protected by an ACL. To help prevent a client from obtaining unauthorized access to information in these files, the server can impersonate the client before accessing the files.

Access Tokens for Impersonation

Access tokens are objects that describe the security context of a process or thread. They provide information that includes the identity of a user account and a subset of the privileges available to the user account. Every process has a primary access token that describes the security context of the user account associated with the process. By default, the system uses the primary token when a thread of the process interacts with a securable object. However, when a thread impersonates a client, the impersonating thread has both a primary access token and an impersonation token. The impersonation token represents the client's security context, and this access token is the one that is used for access checks during impersonation. When impersonation is over, the thread reverts to using only the primary access token. You can use the OpenProcessToken() function to get a handle to the primary token of a process. Use the OpenThreadToken() function to get a handle to the impersonation token of a thread.

Client Impersonation

The Microsoft Windows API provides the following functions to begin an impersonation:

1. A Dynamic Data Exchange (DDE) server application can call the DdeImpersonateClient() function to impersonate a client.
2. A named-pipe server can call the ImpersonateNamedPipeClient() function.
3. You can call the ImpersonateLoggedOnUser() function to impersonate the security context of a logged-on user's access token.

4. The ImpersonateSelf() function enables a thread to generate a copy of its own access token. This is useful when an application needs to change the security context of a single thread. For example, sometimes only one thread of a process needs to enable a privilege.
5. You can call the SetThreadToken() function to cause the target thread to run in the security context of a specified impersonation token.
6. A Microsoft Remote Procedure Call (RPC) server application can call the RpcImpersonateClient() function to impersonate a client.
7. A security package or application server can call the ImpersonateSecurityContext() function to impersonate a client.

For most of these impersonations, the impersonating thread can revert to its own security context by calling the RevertToSelf() function. The exception is the RPC impersonation, in which the RPC server application calls RpcRevertToSelf() or RpcRevertToSelfEx() to revert to its own security context.

Impersonation Levels

If impersonation succeeds, it means that **the client has agreed to let the server "be" the client to some degree**. **The varying degrees of impersonation are called impersonation levels, and they indicate how much authority is given to the server when it is impersonating the client**. Currently, there are four impersonation levels: anonymous, identify, impersonate, and delegate. Prior to Microsoft Windows 2000, the only supported impersonation levels were identified and impersonate. In Windows 2000, delegate-level impersonation is supported. The following list briefly describes each impersonation level.
Anonymous level (RPC_C_IMP_LEVEL_ANONYMOUS) - The client is anonymous to the server. The server process can impersonate the client, but the impersonation token does not contain any information about the client. This level is only supported over the local interprocess communication transport. All other transports silently promote this level to identify.
Identify level (RPC_C_IMP_LEVEL_IDENTIFY) - The system default level. The server can obtain the client's identity, and the server can impersonate the client to do ACL checks.
Impersonate level (RPC_C_IMP_LEVEL_IMPERSONATE) - The server can impersonate the client's security context while acting on behalf of the client. The server can access local resources as the client. If the server is local, it can access network resources as the client. If the server is remote, it can access only resources that are on the same machine as the server.
Delegate level (RPC_C_IMP_LEVEL_DELEGATE) - The most powerful impersonation level. When this level is selected, the server (whether local or remote) can impersonate the client's security context while acting on behalf of the client. During impersonation, the client's credentials (both local and network) can be passed to any number of machines. This level is supported only in Windows 2000 and later versions. For impersonation to work at the delegate level, the following requirements must be met:

1. The client must set the impersonation level to RPC_C_IMP_LEVEL_DELEGATE.
2. The client account must not be marked "Account is sensitive and cannot be delegated" in the Active Directory Service.
3. The server account must be marked with the "Trusted for delegation" attribute in the Active Directory Service.
4. The computers hosting the client, the server, and any "downstream" servers must all be running Windows 2000 in a Windows 2000 domain.
5. By choosing the impersonation level, the client tells the server how far it can go in impersonating the client. The client sets the impersonation level on the proxy it uses to communicate with the server.

Setting the Impersonation Level

There are two ways to set the impersonation level:

1. The client can set it processwide, through a call to CoInitializeSecurity().
2. A client can set proxy-level security on an interface of a remote object through a call to IClientSecurity::SetBlanket() (or the helper function CoSetProxyBlanket()).

You set the impersonation level by passing an appropriate RPC_C_IMP_LEVEL_xxx value to CoInitializeSecurity() or CoSetProxyBlanket() through the dwImpLevel parameter. Different authentication services support delegate-level impersonation to different extents. For instance, NTLMSSP on Windows 2000 supports cross-thread and cross-process delegate-level impersonation, but not cross-machine. On the other hand, the Kerberos protocol (implemented by Windows 2000) supports delegate-level impersonation across machine boundaries, while SChannel does not support any impersonation at the delegate level. If you have a proxy at impersonate level and you want to set the impersonation level to delegate, you should call IClientSecurity::SetBlanket() using the default constants for every parameter except the impersonation level. COM will choose NTLM locally and the Kerberos protocol remotely (when the Kerberos protocol will work).

Registry Key Security and Access Rights

This section just concentrates on the Registry access control. We will learn more detail about registry in another Module. The Windows security model enables you to control access to registry keys. You can specify a security descriptor for a registry key when you call the RegCreateKeyEx() or RegSetKeySecurity() function. If you specify NULL, the key gets a default security descriptor. The ACLs in a default security descriptor for a key are inherited from its direct parent key. To get the security descriptor of a registry key, call the

GetNamedSecurityInfo() or GetSecurityInfo() function. The valid access rights for registry keys include the DELETE, READ_CONTROL, WRITE_DAC, and WRITE_OWNER standard access rights. Registry keys do not support the SYNCHRONIZE standard access right. The following table lists the specific access rights for registry key objects.

Value	Meaning
KEY_ALL_ACCESS	Combines the STANDARD_RIGHTS_REQUIRED, KEY_QUERY_VALUE, KEY_SET_VALUE, KEY_CREATE_SUB_KEY, KEY_ENUMERATE_SUB_KEYS, KEY_NOTIFY, and KEY_CREATE_LINK access rights.
KEY_CREATE_LINK	Reserved for system use.
KEY_CREATE_SUB_KEY	Required to create a subkey of a registry key.
KEY_ENUMERATE_SUB_KEYS	Required to enumerate the subkeys of a registry key.
KEY_EXECUTE	Equivalent to KEY_READ.
KEY_NOTIFY	Required to request change notifications for a registry key or for subkeys of a registry key.
KEY_QUERY_VALUE	Required to query the values of a registry key.
KEY_READ	Combines the STANDARD_RIGHTS_READ, KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY values.
KEY_SET_VALUE	Required to create, delete, or set a registry value.
KEY_WOW64_64KEY	Enables a 64- or 32-bit application to open a 64-bit key on 64-bit Windows. This flag must be combined using the OR operator with the other flags in this table that either query or access registry values.
KEY_WOW64_32KEY	Enables a 64- or 32-bit application to open a 32-bit key on 64-bit Windows. This flag must be combined using the OR operator with the other flags in this table that either query or access registry values.
KEY_WRITE	Combines the STANDARD_RIGHTS_WRITE, KEY_SET_VALUE, and KEY_CREATE_SUB_KEY access rights.

Table 19

When you call the RegOpenKeyEx() function, the system checks the requested access rights against the key's security descriptor. If the user does not have the correct access to the registry key, the open operation fails. If an administrator needs access to the key, the solution is to enable the SE_TAKE_OWNERSHIP_NAME privilege and open the registry key with WRITE_OWNER access. You can request the ACCESS_SYSTEM_SECURITY access right to a registry key if you want to read or write the key's SACL.