Global Information Assurance Certification Paper

Windows 2000 Access Control Lists – A peek under the hood.
GSEC Practical Assignment v.1.4b

Ahmed Farouk AbdulAzim
December 27, 2002

Abstract

Access Control Lists is an extremely important element of the Access Control Model, and understanding it well helps administrators to set correct and more secure permissions on resources, and to understand the weakness and default behavior of Windows 2000. In-depth knowledge of the operating system is the "basic" tool to maintain more secure networks. One of the immutable laws of security administration is "The most secure network is a well-administered one" [1]. This law can't be observed without "looking under the hood" of the operating system. This paper provides the reader with a look at a critical part of every Windows 2000 based computer, Access Control Lists.

What are Access Control Lists?

Since Windows 2000 supports C2-Level security as defined by the US Department of Defense "Orange Book" [2], it is mandated that the operating system meet certain requirements, of which the following directly relate to the subject of this paper:

1. Access to a resource, be it granting or denying, to a user or group of users, should be possible to control.
2. It should be possible to audit security related events.

Both the above requirements are implemented in the Windows NT/2000/XP Access Control Model. This model is composed of various components and tasks, of which Access Control Lists (ACL) is a key component. An ACL, in most cases, is attached to a securable object, and contains a list of who is allowed access to the object, who is denied access to the object, what level of access is either allowed or denied, and if any security related action regarding the object should be audited in a security log. A securable object is any object that could have a security descriptor, which in turn is a structure that contains certain security related information. An ACL is part of this security descriptor that is attached to securable objects. Examples of securable objects range from named objects such as files, folders, Active Directory objects and registry keys, as well as non-named objects such as processes and threads.

Components of Access Control Model

The two main components of the Access Control Model are:

1. Access Tokens
2. Security Descriptors.

Access tokens provide the security context that users and processes use to interact with securable objects. They are created by the Local Security Authority when authentication of a user login is successful. Any process running on behalf of the user carries a copy of the user's access token. The access token is composed of the following components:

- User SID. This is the Security Identifier of the user, and is unique within the computer (non domain) or throughout Active Directory (domain model). Windows 2000 refers to users and groups by their SIDs rather than by their names.
- Group SIDs. A list of the SIDs of the groups that the user is a member of.
- Privileges or rights held on the local computer by the user and the groups that the user is a member of. Rights can be modified through the Local Security MMC snap-in. Several security rights exist such as the right to take ownership of files and other objects, which is by default given to the Administrators group.
- Primary group that the user is a member of. This is only used by the POSIX subsystem of Windows 2000 and is otherwise ignored.
- Default Discretionary ACL. This ACL is used in the creation of new objects by the user when no other ACL is defined or available. By default, this ACL gives Creator Owner (user who created the object) and System (the local computer system) full control over the newly created object.
- Source process that created the access token. This could be for example the LAN Manager or the RPC Server.
- Type of token. A token could be of two types, primary or impersonation. Primary tokens represent the security context of the user. While an impersonation token could be used by a thread of a service to temporarily adapt a different security context than the parent process. Impersonation is especially useful in client/server operations.
- Impersonation Level. Various levels of impersonation exist, and this field provides a value of to which level can a service adopt the security context of a user. This level can range from Anonymous, being the lowest level, up to Delegate, being the highest level. Anonymous is where the service can impersonate the user, but the impersonation token contains no information about the user. While Delegate is a level where the service can impersonate the user when accessing resources on the same computer as the service, or any resource on any other computer.
- Statistics about the access token itself which Windows uses internally.
- Restricting SIDs, which is an optional list of SIDs that is attached to the token by a process that has the authority to create a restricted token, which can be used to give a thread a lower security context to run in than the user.
- Session ID. This value is used to indicate if the token is associated with a Terminal Services session.

Security Descriptor, which is a structure containing security related information, and which is attached to securable objects. Parts of this data structure include:

- Header, which contains information such as a revision number, memory layout, and which elements are present.
- Owner SID. This is the SID of the owner of this object.
- Primary group SID. As in access tokens, this is the SID of the primary group that the owner of the object belongs to.
- Discretionary ACL (DACL). This is a list usually containing several Access Control Entries (ACE). An ACE is a structure containing information regarding granting or denying access to a user or group as well as the level of access.
- System ACL (SACL). Similar to the DACL structure. SACLs also contain ACEs but rather than containing information on granting or denying access, they contain information on what action to audit, type of action (success or failure), and which user or group to audit. In addition, SACLs can only be controlled by users or groups that have the privilege to manage audit and security logs, which by default is only given to the Administrators group.
- Layout in memory. This could be either self-relative or absolute, and a flag in the header part indicates which layout it is.

Components of an ACL

Both DACLs and SACLs have identical format on the higher level. ACLs are composed of:

1. An ACL structure that contains various information about the ACL including:
    a. ACL Revision. The ACL structure for all revisions is the same, but what might vary is the structure of ACEs. This is used in cases when an object specific ACE exists. An example of this is Active Directory objects.
    b. ACL size. This includes the size of the header plus ACEs.
    c. ACE Count. This is the number of ACEs that exist in the ACL.

2. ACEs. In the case of DACLs, ACEs are the specific entries that determine which users and groups have access to the object, and what level of access is granted or denied. In the case of SACLs, ACEs provide the specific entries that determine what users or groups to audit, what actions to audit, and if the audit trigger is a success or failure action.

In Windows 2000, eight types of ACEs exist, two of which are reserved for future use with SACLs. The remaining six could be divided into generic ACEs and object specific (Active Directory) ACEs. Object specific ACEs build upon the same structure as generic ACEs but also adds three more fields to the structure. The basic fields present in both generic and object specific ACEs are:

- ACE Header. This header contains information about the type of ACE and its size. It is made up of three parts:
  - ACE Type. This could be one of six values, depending on if the ACE is generic or object specific, if the ACE is an allowed or denied ACE, and if the ACE is for a SACL (Appendix, Table 1).
  - ACE Flags. This field specifies if the ACE should be inherited or not, whether it is effective on the current container and in the case of SACLs if the ACE is a successful or failure attempt ACE (Appendix, Table 2).
  - ACE size in bytes.
- Access Mask. This field is a double-word structure containing a value that maps to a right that's allowed, denied or audited. For instance, if the "write" bit is set, and the ACE is an "allow" ACE, the access mask would "allow write" (Appendix, Table 5).
- SID of the user or group that the ACE controls access for, or audits.

If the ACE is an object specific (Active Directory) ACE, three other fields exist. These fields offer a more fine-tuned control over Active Directory objects. Only two types of generic objects exist, containers and non-containers. Container objects are ones that can contain other container or non-container objects. While non-containers are objects that cannot contain any objects. Simple examples of generic objects are folders and files. Folders are container objects; they could contain other sub-folders or files. Files are non-container objects that cannot contain any other objects. Generic ACEs can differentiate between container and non-container objects only. Active Directory holds many different types of objects with different attributes. For example an Organizational Unit (OU) is a container object that can contain a non-container object such as a User and a Computer object. Object specific ACEs could be attached to the OU, and allowed to be inherited only by Computer objects. Also object specific ACE can grant or deny access to a specific property or property set of an object. For example, a Domain Administrator might give a user the right to edit his own telephone number field in Active Directory (this is a property), or give the user the right to change all personal information (this is a property set) that included address, telephone, fax, etc. This is an example of the granularity of access that object specific ACEs give, and I have found out by experience that this is extremely beneficial in delegating administrative tasks.

The three object specific fields are:

- Flags. These flags tell if the following two fields are present in the ACE or not. The following two fields govern what type of object is controlled by the ACE, and what type of object should inherit the ACE.
- Object Type. Five types of object types exist as in Appendix Table 3.
- Inherited Object Type. This field specifies which type of child object can inherit the ACE. This builds over and above the normal container and non-container inheritance flags in generic ACEs.

Objects, properties, and property sets in Active Directory are referenced by a Globally Unique Identifier (GUID). GUID is Microsoft's implementation of the Universal Unique Identifier (UUID) which is used in the Open Software Foundation's Distributed Computing Environment (DCE). UUIDs are 128 bit long unique numbers. The uniqueness of UUIDs is guaranteed by combining the hardware (MAC) address of the network card of the host that generated the UUID with a time stamp, and applying random seeds [12]. SIDs are still used by newer Microsoft Windows products such as Windows 2000 and XP only for backward compatibility, and is suppose to be phased out some time in the future.

ACLs in action

Now that we have covered the elements of ACLs as well as other components of the Access Control Model, let us look at how these components and elements interact together to maintain access control.

Objects are controlled by object managers. Each object type has a different object manager that controls access to the object, and provides default security permissions when no other permissions are available. A list of object types and object managers is available in Appendix Table 4.

When a process or process thread tries to access an object, the object's manager calls the API function AccessCheckAndAuditAlarm, which handles the functionality of checking the object access rights and either granting or denying access. This function takes in consideration three types of access masks:

- Requested Access Mask: This maps to the rights that the thread or process want to gain and carry out on the object, for instance "Read".
- Object Access Mask: Each ACE in the objects ACL has an access mask that maps to the right the ACE is allowing, denying or auditing.
- Granted Access Mask: This is initially set to zero, i.e. an empty access mask, and as ACEs are reviewed against the requested access mask, matching bits are set in the granted access mask.

After reviewing all ACEs in the object ACL, AccessCheckAndAuditAlarm returns either an empty granted access mask, which denies access to the object, or a granted access mask that's exactly the same as the requested access mask, which grants access.

Before explaining in details how AccessCheckAndAuditAlarm reviews ACLs, it is important to understand that an object could have an empty ACL or no ACL at all. An empty ACL is one that contains no ACEs. This would deny everyone access to the object, including the object owner and Administrators, however Administrators and object owners can still gain access to the object by modifying the ACL and adding an allow ACE. On the other hand, null ACLs (non existent) would allow everyone full control to the object. Null ACLs is a rare case that could occur if during the object's creation no ACL was provided by the creating process, nor was a default ACL found. Object creation and how ACLs are attached to newly created objects is covered later on in this paper.

In details, AccessCheckAndAuditAlarm does the following:

1. If the object's security descriptor contains no ACL, the granted access mask is set to match the requested access mask, and the granted access mask is returned to the object manager, effectively giving the calling thread the rights it requested to the object.
2. If the object's security descriptor does contain an ACL, the ACE Count field is checked. If this field equals to zero, this means that no ACEs exist in the ACL, i.e. denying access to everyone. The granted access mask is all set to zero, denying access, and is returned to the object manager. By design, this step comes before the owner of the object is checked; hence this is why even the object owner is denied access to the object.
3. The requested access mask is reviewed, if it is empty, this means that the requesting thread did not specify what level of access is required, and hence AccessCheckAndAuditAlarm sets the granted access mask to zero, denying access, and returns it to the object manager.
4. If the AS bit is set in the requested access mask, the privileges field of the requestor access token is reviewed. If this field contains the privilege to manage auditing and security log, the AS bit in the granted access mask is also set. The corresponding AS bit is reset in the requested access mask. If the requested access mask becomes all zeros, access checking stops.
5. If the change permissions, read permissions or modify owner bits are set in the requested access mask, the user and group SID of the requestor are compared to the object owner user and group SID. If a match exists, the bit is turned on in the granted access mask, and turned off in the requested access mask. If any bits in the requested access mask are still on, access checking continues, or else it ends.
6. A counter is set to equal ACE Count. This is used to go through each ACE in the ACL.
7. If the counter equals (ACE Count + 1), the granted access mask is all set to zeros and access checking stops. This effectively denies access to the object. This is because the end of the ACL was reached yet not all the requested permissions were granted. By design when such a situation is

encountered the system revokes any bits granted in the granted access mask and ends access checking.

8. ACE[Counter] is reviewed, if the INHERIT_ONLY flag is set this means that ACE is not an effective ACE and is only present for inheritance, hence the ACE is skipped. The counter is incremented and the function returns to step 7.
9. The access token user and group SIDs are compared against the ACE user and group SIDs. If the SIDs don't match, the ACE is skipped and the function returns to step 7 after incrementing the counter. This is because the ACE does not apply to the current user/group and hence the function need not review it.
10. If the ACE is a deny ACE, the requested and ACE access masks are compared. If any bit is set on both masks, the granted access mask is reset to zeros and access checking stops, denying access. This is because the requestor tried to gain a permission that was denied, and hence all granted access was revoked. If no matches exist, the function increments the counter and goes back to step 7. This is because even though the requestor was denied a certain permission, but the requestor did not try to gain that specifically denied permission, and hence access checking continues.
11. If the ACE is an allow ACE, the ACE and requestor access masks are compared. If a match exists, the matching bit is turned on in the granted access mask, and off in the requested access mask. If the requested access mask is all zeros, then all requested permissions have been granted, hence the there is no need to continue in the ACL, and access checking stops. If however the requested access mask is not all zeros, the counter is incremented and the function returns to step 7, to review the rest of the ACL.

Flow chart 1 in the appendix shows the steps carried out when AccessCheckAndAuditAlarm reviews an object's DACLs.

The steps above explain how DACLs are reviewed. SACLs differ in the way they are checked. Auditing comes after access check is completed and the granted access mask returned to the object manager. This is because the system needs to audit actions *after* they are taken. It is important to note that when auditing the function does not terminate until all ACEs are reviewed. Below are the detailed steps of auditing and reviewing an object's SACL:

1. The SACLs ACE Count field is checked. If it equals to zero, then the SACLs is empty and there is no need to continue since no auditing entries are attached to the object. The function ends.
2. A counter is set to equal the ACE Count. This counter is used to pass through all the ACEs in the SACL.
3. If the counter is equal to (ACE Count + 1), the function ends. This means that the end of the SACL has been reached, hence auditing is stopped.

4. ACE[Counter] is reviewed, if the INHERIT_ONLY flag is set, there is no need to review the rest of the ACE since this ACE is non-effective on the current object and is only kept to be inherited by child objects. The counter is incremented and the function returns to step 3.
5. The user and group SIDs of the thread's access token is compared to the ACE[Counter] SID. If they do not match, this means that this ACE is not relevant to the current user and group, hence is skipped by incrementing the counter and returning to step 3.
6. If the user and/or group SIDs of the thread's access token match those in the ACE[Counter], the ACE[Counter] access mask is reviewed against the requested access mask. If a bit is turned on in the ACE access mask and the same bit is turned off in the requested access mask this means that the permission requested by the thread is not audited by the current ACE. The counter is incremented and the function returns to step 3.
7. The ACE is reviewed to check what type of system ACE it is (success or failure). If the flag SUCCESSFULL_ACCESS_ACE_FLAG is set, the ACE access mask is compared to the granted access mask. If a bit is set on both, this means that the thread did gain access to the object. The function would record a successful entry in the security log, and will consider information from the thread's access token (who triggered the event?) as well as the permission corresponding to the matching bit (what action was audited). The counter is incremented and the function returns to step 3.
8. If the flag FAILED_ACCESS_ACE_FLAG is set in the current ACE, the ACE access mask is compared to the granted access mask. If a bit is turned on in the ACE access mask and the same bit is turned off in the granted access mask this means that the object failed to gain the specific permission to the object. A failure event is logged in the security log, with relevant information from the thread's access token and the ACE access mask. The counter is incremented and the function returns to step 3.

Flow chart 2 in the appendix shows the steps carried out when AccessCheckAndAuditAlarm reviews an object's SACLs.

Creation of ACLs

Part of the creation of an object, is the creation of the security descriptor that attaches to the newly created object. The information needed in the constructing of the security descriptor include object owner, ACL, as well as inheritance properties. The sources used to collect this information include one or more of the following:

- Subject, which is the process or thread that created the object. The subject's access token would provide information such as owner SID and default ACL. The owner SID of the access token becomes the owner of

the newly created object. The ACL is constructed from the default ACL as well as any inheritable ACEs as we will see later.

- Object Managers. As explained earlier, different types of objects are controlled by different object managers (Appendix Table 4). The object manager of the newly created object is referenced to provide an ACL if none are provided by the subject or through inheritance.
- Parent Objects. The newly created object can inherit the ACL from its parent. For example a new file created within a folder, could inherit the ACL of the folder.

It is quite easy to see the manipulation of ACLs through the NTFS file system. The creator of a folder could for example block the inheritance of permissions from the parent folder. Or the administrator could allow the inheritance of permissions from a parent, and then stop the child object from receiving any changes in the parent permissions. Permissions could be set to be inheritable or not, and could also be tuned to be inherited by containers or non-containers.

It is important to note that permissions could be inherited or explicit. A sub-folder could have all its ACL inherited from the parent folder, as well as having ACEs that are applied directly on the sub-folder. An extremely important and yet dangerous default behavior in Windows 2000 and XP puts explicit ACEs ahead of inherited ACEs. As any Windows 2000 administrator knows, a deny entry overrides any other allowed entry. This is because of the way ACEs are sorted in a DACL. However the Windows interface hides certain aspects that could be manipulated using APIs. For instance, the Windows interface puts deny ACEs ahead of allow ACEs, but if the ACL was constructed through a program, it could sort ACEs as it wants, and override the default behavior of deny ACEs coming before allow ACEs. Even though that breaking the default behavior is not recommended by Microsoft, it is quite easy for anyone with programming knowledge and the correct APIs to override this behavior.

All explicit ACEs come before inherited ACEs. This would mean that an explicit allow ACE supersedes an inherited deny ACE. An example of this could be seen in Image 1 in Appendix. In this example I created a folder and named it “Test Folder” and removed all inherited permissions from the parent. I created a sub folder “Sub Folder” under the “Test Folder”. On “Test Folder” I denied access to user Farouk, and at the same time allowed the same user access on “Sub folder”. Image 1 shows the order that the ACEs are reviewed by the function AccessCheckAndAuditAlarm for “Sub Folder”. The bottom two ACEs are inherited ones. The top ACE is the explicit ACE. Even though Farouk is denied access to “Sub Folder” at the parent “Test Folder” level, if the user typed the full path of the folder (example C:\Test Folder\Sub Folder\) he will gain instant access.

When permissions on a parent folder are changed, the system automatically propagates the changed permissions to children objects, but according to

inheritance rules (a child object might be configured to block inheritance of permissions). This does not remove explicit permissions. If it is required to remove explicit permissions, the option “Reset permissions on all child objects” would force all child ACLs to be removed and replaced by inherited ACLs from the parent.

In Active Directory the construction of ACLs differs in two ways:

- The Active Directory schema could provide a security descriptor for the created object. This is because the schema stores a default security descriptor attribute for every object class defined.
- Since ACE could be object-specific, these object specific ACE are only inherited by the specified object types. This is over and above the two generic types of objects (container and non-container).

Conclusion

Access Control is a critical aspect of any operating system. Understanding Access Control Lists in Windows 2000 and its default behavior greatly helps administrators to set correct resource access control, and safe guard against mis-configurations.

References

[1] Culp, Scott. “The Ten Immutable Laws of Security Administration”. Microsoft Security Essays. November 2000. URL: http://www.microsoft.com/technet/columns/security/essays/10salaws.asp

[2] US Department Of Defense. “Department Of Defense Trusted Computer System Evaluation Criteria”. Department Of Defense Standard. December 1985. URL: http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html.

[3] Microsoft Corp. “Access Control”. Windows 2000 Server Distributed Systems Guide. URL: http://www.microsoft.com/technet/prodtechnol/windows2000serv/reskit/distsys/part2/dsgch12.asp

[4] Microsoft Corp. “Access Control Components”. Platform SDK. October 2002. URL: http://msdn.microsoft.com/library/en-us/security/security/access_control_components.asp

[5] Microsoft Corp. “Access Tokens”. Platform SDK. October 2002. URL: http://msdn.microsoft.com/library/en-us/security/security/access_tokens.asp

[6] Microsoft Corp. "Security Descriptors". Platform SDK. October 2002. URL: http://msdn.microsoft.com/library/en-us/security/security/security_descriptors.asp

[7] Microsoft Corp. "ACL". Platform SDK. October 2002. URL: http://msdn.microsoft.com/library/en-us/security/security/acl.asp

[8] Microsoft Corp. "ACE". Platform SDK. October 2002. URL: http://msdn.microsoft.com/library/en-us/security/security/ace.asp

[9] Microsoft Corp. "ACE_HEADER". Platform SDK. October 2002. URL: http://msdn.microsoft.com/library/en-us/security/security/ace_header.asp

[10] Microsoft Corp. "Object-Specific ACEs". Platform SDK. October 2002. URL: http://msdn.microsoft.com/library/en-us/security/security/object_specific_aces.asp

[11] Microsoft Corp. "ACCESS_ALLOWED_OBJECT_ACE". Platform SDK. October 2002. URL: http://msdn.microsoft.com/en-us/security/security/access_allowed_object_ace.asp

[12] Microsoft Corp. "AccessCheckAndAuditAlarm". Platform SDK. October 2002. URL: http://msdn.microsoft.com/library/en-us/security/security/accesscheckandauditalarm.asp

[13] Microsoft Corp. "Understanding Container Access Inheritance Flags in Windows 2000". Microsoft Knowledge Base. October 10, 2002. URL: http://support.microsoft.com/default.aspx?scid=KB;EN-US;Q220167&

[14] Zenkov, Igor P. "COM+ Glossary". URL: http://www.geocities.com/izenkov/Ref/Com-uuid.htm

[15] Pedestal Software. "Windows 2000 Access Control". NTSEC Security Tools. URL: http://www.pedestalsoftware.com/ntsec/ntsecfaq2000.htm

[16] Microsoft Corp. "Objects and Object Managers". Windows 2000 Server Help.

[17] Microsoft Corp. "Microsoft Official Curriculum: Designing a Secure Windows 2000 Network - Workbook". Microsoft Corp. June 2000.

Appendix

Table 1 ACE Types [8]

Value	Description
ACCESS_ALLOWED_ACE_TYPE	Generic ACE allowing access.
ACCESS_DENIED_ACE_TYPE	Generic ACE denying access.
SYSTEM_AUDIT_ACE_TYPE	Generic ACE for SACLs.
ACCESS_ALLOWED_OBJECT_ACE_TYPE	Object specific ACE allowing access.
ACCESS_DENIED_OBJECT_ACE_TYPE	Object specific ACE denying access.
SYSTEM_AUDIT_OBJECT_ACE_TYPE	Object specific ACE for SACLs.

Table 2 ACE Flags [9]

Value	Description
CONTAINER_INHERIT_ACE	Child container objects inherit this ACE as an effective ACE.
OBJECT_INHERIT_ACE	Child non-container objects inherit this ACE as an effective ACE. If the child object is a container object, it will also inherit it (to pass it to grandchildren) but it will not be effective on the non-container.
INHERIT_ONLY_ACE	Indicates that the ACE is not effective on the current object and is present only to be inherited.
NO_PROPAGATE_INHERIT_ACE	If this flag is set, the ACE is inherited by the child object, but on the child object the ACE flags that allow the ACE to be inherited are cleared. This prevents the next level of objects (grandchildren) from inheriting the ACE.
INHERITED_ACE	The ACE is inherited. The system sets this flag when propagating an ACE.
SUCCESSFUL_ACCESS_ACE_FLAG	Used in SACLS to audit success operations on the object that the ACE is attached to.

FAILED_ACCESS_ACE_FLAG	Used in SACLS to audit failure operations on the object that the ACE is attached to.

Table 3 ACE Object Types [11]

Value	Description
ADS_RIGHT_DS_CREATE_CHILD	The ACE controls if the user or group can create a specific type of child object.
ADS_RIGHT_DS_READ_PROP	The ACE controls if the user or group can read a specific property or property set.
ADS_RIGHT_DS_WRITE_PROP	The ACE controls if the user or group can write a specific property or property set.
ADS_RIGHT_DS_CONTROL_ACCESS	The ACE controls if the user or group can perform a specific extended right on the object. Extended rights are ones that are not covered by the standard access rights. For example the right to send mail on another user's behalf is an extended right.
ADS_RIGHT_DS_SELF	The ACE controls if the user or group can perform a special validated write operation. A validated write is different from a normal write operation because it requires that the system check that the value to be written is valid, for example within a certain range.

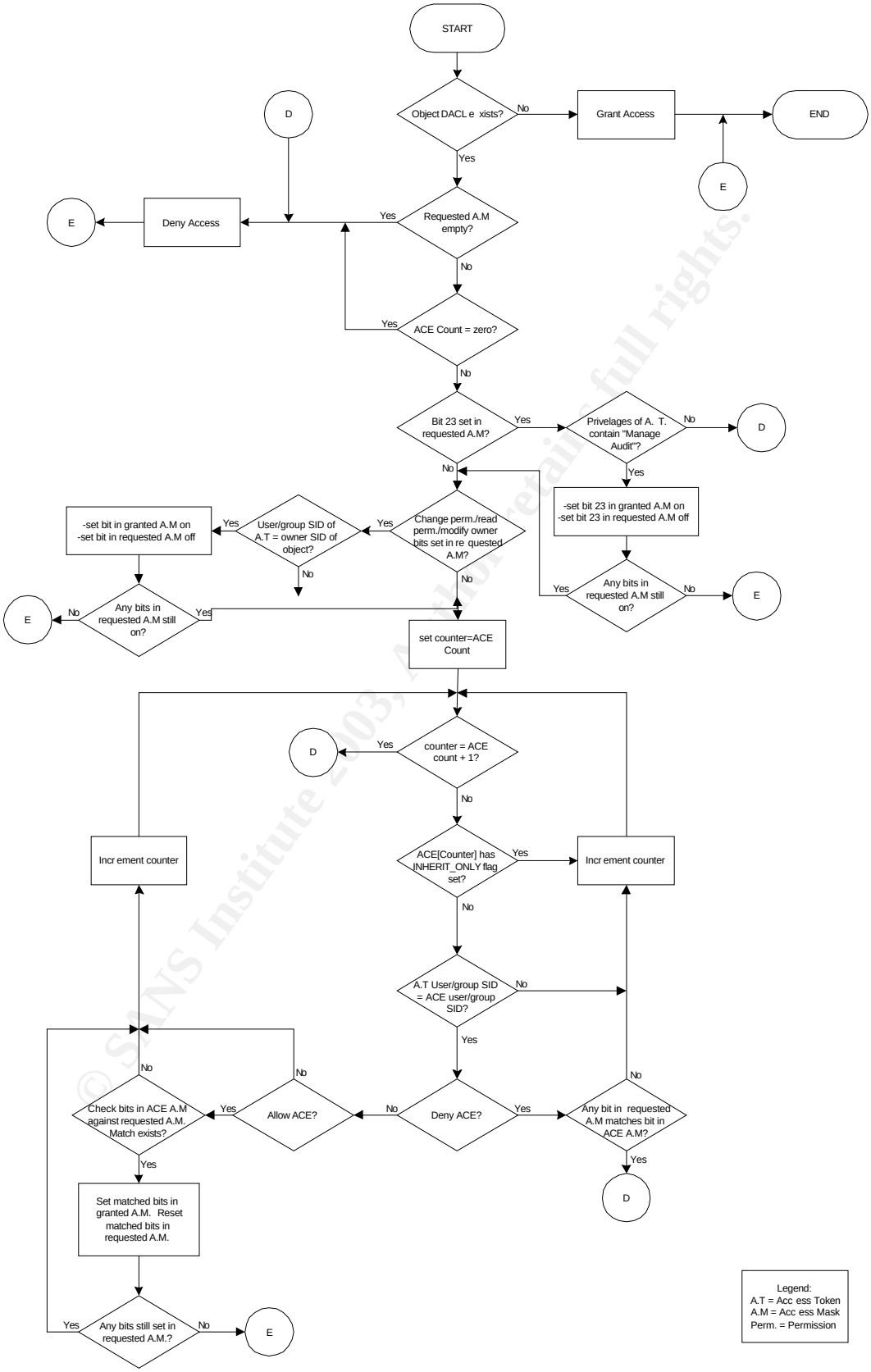
Table 4 Object Managers [14]

Object type	Object manager
Files & Folders	NTFS file system
Active Directory objects	Active Directory
Shares	Server service
Registry keys	The registry
Services	Service controllers
Printer	Printer spooler

Table 5 Access Mask format [3]

Bit number	Corresponding permission
0 – 15	Object-specific access rights
16 – 22	Standard access rights
23	Right to access the SACL.
24 – 27	Reserved for future use.
28	Generic All (read, write & execute) rights.
29	Generic execute.
30	Generic write.
31	Generic read.

Flowchart 1 DACL in action



Flowchart 2 SACL in action

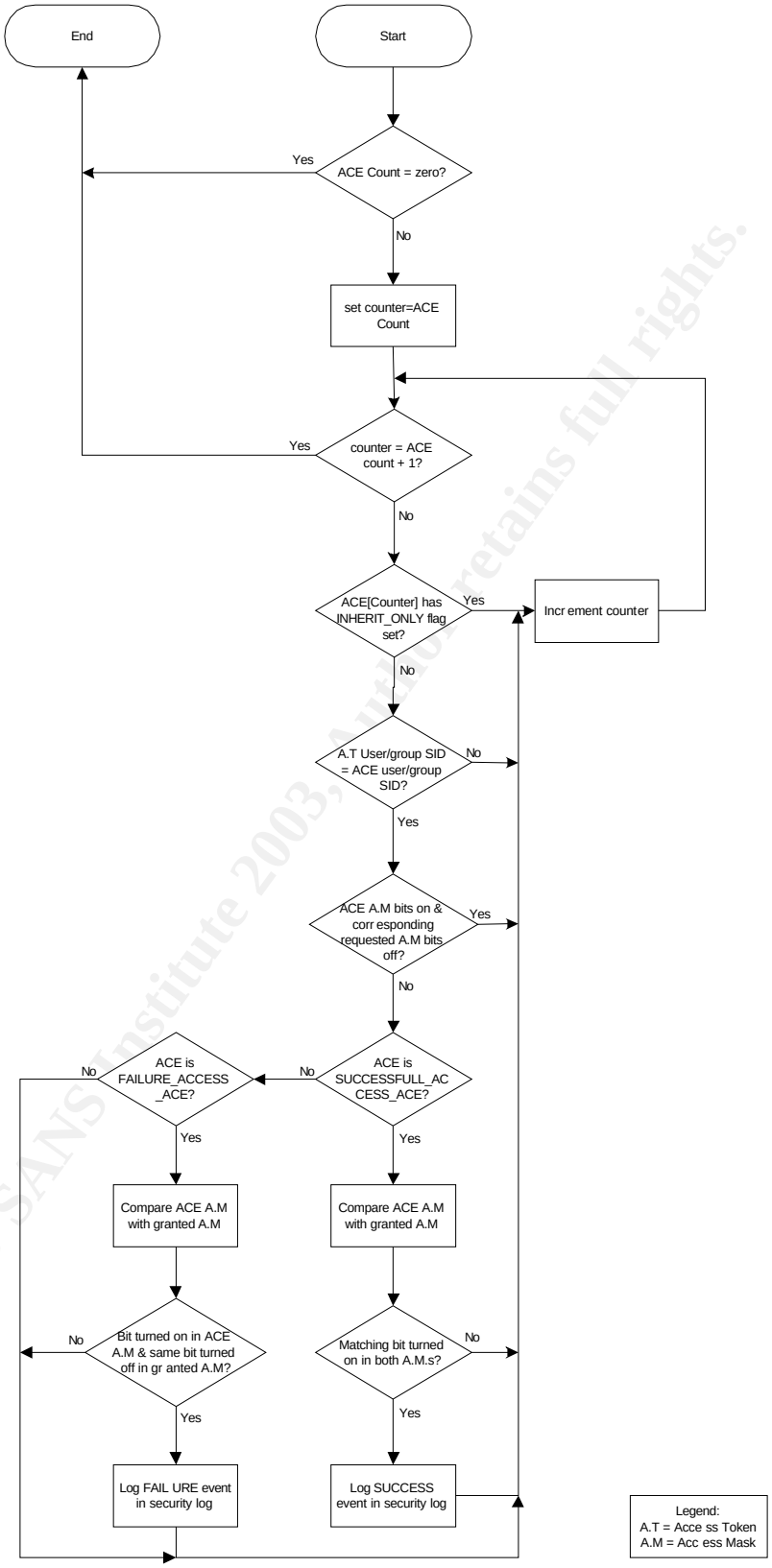
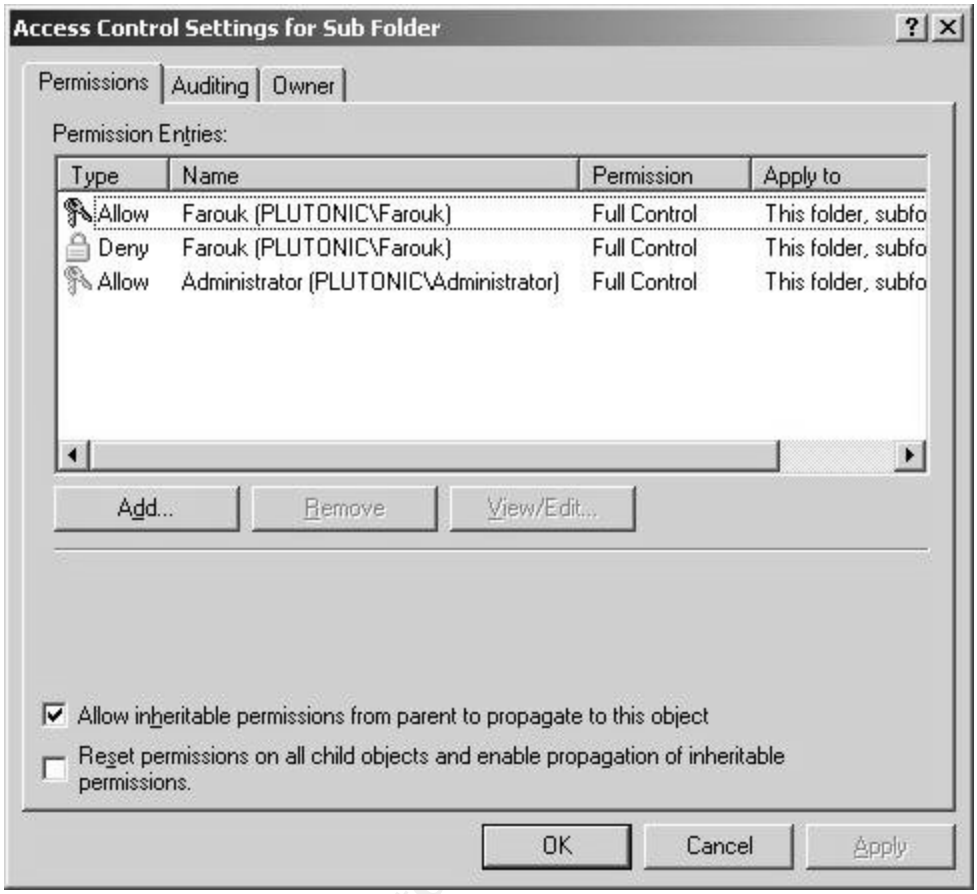


Image 1 Explicit vs. inherited ACEs.



Last Updated: May 28th, 2017

Upcoming Training

SANS Stockholm 2017	Stockholm, Sweden	May 29, 2017 - Jun 03, 2017	Live Event
SANS San Francisco Summer 2017	San Francisco, CA	Jun 05, 2017 - Jun 10, 2017	Live Event
SANS Houston 2017	Houston, TX	Jun 05, 2017 - Jun 10, 2017	Live Event
Security Operations Center Summit & Training	Washington, DC	Jun 05, 2017 - Jun 12, 2017	Live Event
Community SANS Ottawa SEC401	Ottawa, ON	Jun 05, 2017 - Jun 10, 2017	Community SANS
SANS Charlotte 2017	Charlotte, NC	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Rocky Mountain 2017 - SEC401: Security Essentials Bootcamp Style	Denver, CO	Jun 12, 2017 - Jun 17, 2017	vLive
Community SANS Portland SEC401	Portland, OR	Jun 12, 2017 - Jun 17, 2017	Community SANS
SANS Secure Europe 2017	Amsterdam, Netherlands	Jun 12, 2017 - Jun 20, 2017	Live Event
SANS Rocky Mountain 2017	Denver, CO	Jun 12, 2017 - Jun 17, 2017	Live Event
SANS Minneapolis 2017	Minneapolis, MN	Jun 19, 2017 - Jun 24, 2017	Live Event
SANS Columbia, MD 2017	Columbia, MD	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS Cyber Defence Canberra 2017	Canberra, Australia	Jun 26, 2017 - Jul 08, 2017	Live Event
SANS Paris 2017	Paris, France	Jun 26, 2017 - Jul 01, 2017	Live Event
SANS London July 2017	London, United Kingdom	Jul 03, 2017 - Jul 08, 2017	Live Event
Cyber Defence Japan 2017	Tokyo, Japan	Jul 05, 2017 - Jul 15, 2017	Live Event
SANS Cyber Defence Singapore 2017	Singapore, Singapore	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Minneapolis SEC401	Minneapolis, MN	Jul 10, 2017 - Jul 15, 2017	Community SANS
SANS Los Angeles - Long Beach 2017	Long Beach, CA	Jul 10, 2017 - Jul 15, 2017	Live Event
Community SANS Phoenix SEC401	Phoenix, AZ	Jul 10, 2017 - Jul 15, 2017	Community SANS
SANS Munich Summer 2017	Munich, Germany	Jul 10, 2017 - Jul 15, 2017	Live Event
Mentor Session - SEC401	Ventura, CA	Jul 12, 2017 - Sep 13, 2017	Mentor
Mentor Session - SEC401	Macon, GA	Jul 12, 2017 - Aug 23, 2017	Mentor
Community SANS Atlanta SEC401	Atlanta, GA	Jul 17, 2017 - Jul 22, 2017	Community SANS
Community SANS Colorado Springs SEC401	Colorado Springs, CO	Jul 17, 2017 - Jul 22, 2017	Community SANS
SANSFIRE 2017	Washington, DC	Jul 22, 2017 - Jul 29, 2017	Live Event
SANSFIRE 2017 - SEC401: Security Essentials Bootcamp Style	Washington, DC	Jul 24, 2017 - Jul 29, 2017	vLive
Community SANS Charleston SEC401	Charleston, SC	Jul 24, 2017 - Jul 29, 2017	Community SANS
Community SANS Fort Lauderdale SEC401	Fort Lauderdale, FL	Jul 31, 2017 - Aug 05, 2017	Community SANS
SANS San Antonio 2017	San Antonio, TX	Aug 06, 2017 - Aug 11, 2017	Live Event
SANS Prague 2017	Prague, Czech Republic	Aug 07, 2017 - Aug 12, 2017	Live Event