Improving the Granularity of Access Control for Windows 2000

MICHAEL M. SWIFT[*], PETER BRUNDRETT, CLIFF VAN DYKE, PRAERIT GARG, ANNE HOPKINS, SHANNON CHAN, MARIO GOERTZEL AND GREGORY JENSENWORTH

Microsoft Corporation

This paper presents the mechanisms in Windows 2000 that enable fine-grained and centrally managed access control for both operating system components and applications. These features were added during the transition from Windows NT 4.0 to support the Active Directory, a new feature in Windows 2000, and to protect computers connected to the Internet. While the access control mechanisms in Windows NT are suitable for file systems and applications with simple requirements, they fall short of the needs of applications with complex data objects. Our goal was to use operating system access control mechanisms to protect a large object hierarchy with many types of objects, each with many data properties. We also wanted to reduce the exposure of users to untrustworthy or exploited programs.

We introduced three extensions to support these goals. First, we extended the entries in access control lists to provide an unlimited number of access rights for a single object and to allow grouping those rights for efficiency. Second, we extended the entries to specify precisely how access control lists are assigned to each distinct type of object, instead of treating all types identically. Finally, we extended the data structure identifying users' identity to the operating system to allow users to restrict the set of objects a program may access. These changes allow a single access control mechanism to be used to protect both system and application resources, as well as protect users from each other and users from their programs, simplifying both program development and system management.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection – *access controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection – *invasive software*
General Terms: Security, Design, Performance
Additional Key Words and Phrases: Access control lists, Microsoft Windows 2000, Windows NT, Active Directory

1. INTRODUCTION

One goal of Windows NT 4.0 operating system was to provide a secure platform for applications by providing general support for authentication, access control, and auditing. However, the addition of the Active Directory in Windows 2000, the follow-on operating system to Windows NT, and the increasing frequency of security attacks on trusted applications demonstrated several limitations of Windows NT access control. The Active Directory, a hierarchical directory service [Iseminger 2000], requires access control at a finer granularity and with more centralized control than can be supported by the mechanisms in Window NT. The security attacks demonstrated that users could not prevent untrusted code from accessing their data. This paper presents the changes made to the Windows NT access control mechanisms to address these limitations.

The increasing integration of applications, when several independent programs cooperate and share data, drives the need for new access control mechanisms. There are many situations where data from one application must be available to another application,

Originally published in the Proceedings of the 6th ACM Symposium on Access Control Models and Technologies (SACMAT '01, Chantilly, VA, May 2001).
[*] Now at the University of Washington

although not always with the same access rights. For example, an electronic mail server may access a user account database for determining valid email addresses, but should not be able to modify the address. This increased sharing between applications has led to centralized repositories of application data, such as the configuration registry in Windows NT and Windows 95, and directories services such as Novell NDS and Microsoft's Active Directory. These centralized repositories of data require fine-grained protection to restrict each application to only its required access. For example, a mail server may need to modify mail routing information on a user object, but should not be able to set users' passwords.

In many operating systems access control mechanisms are separate for each application, such as permissions in the file system and configuration files for user applications. Windows NT integrates many security services that were formerly provided by applications, such as authentication and access control. As a result, Windows NT has a single access control mechanism that is used by all system components, including kernel objects, user interface objects, and the file system. The access control mechanism is also intended for use by applications, such as web servers or mail servers. This approach benefits both administrators and developers by requiring that they learn only a single set of mechanisms, enables a common user interface for access control, and reduces the amount of security-critical code in applications.

The major access control mechanism in Windows NT is the access control list (ACL), which for each object specifies the operations users may perform. The access control lists in Windows NT 4.0 were designed for services, such as the file system, with only a few types of objects and with only a small number of operations. However, the Active Directory stores data for many different uses, such as logon and authorization, electronic mail, and security policy. It contains hundreds of types of data objects, ranging from user account data to network printer configuration data, and every object has many properties, such as user name and password or printer description. Depending on the needs of the application, the properties on a single object may be protected separately, so some are accessible to all users while other properties are accessible only to administrators. While implementing the directory service we discovered two limitations of the access control mechanisms in Windows NT: the access control lists cannot distinguish between large numbers of operations on a single object or large numbers of types of objects, and cannot propagate access control changes through a tree of objects.

The solution in Windows 2000 to both limitations is to annotate ACLs with additional information, such as whether entries apply to all objects or just a particular type of object. For example, access control lists in Windows 2000 specify which types of objects may be created and deleted, rather than granting the right to create all types of objects. Access control lists also specify which types of objects inherit access control from a container, instead of inheriting the same access control onto all objects. As a result, access control lists can precisely specify users' access to a specific object as well as propagate that access to all objects of the same type.

Another limitation of the access control mechanisms in Windows NT is that they were not designed to protect the users from their programs. The mechanisms assume that users are in control of the code they execute, and provide no features to ensure that programs don't accidentally corrupt or misuse users' data. Unfortunately, it has become increasingly common for users' applications to turn malicious. For example, bugs in trusted applications, such as web browsers and email clients, cause damage by inadvertently exposing all the files on a computer to an attacker [CERT Coordination Center 1995]. Downloaded ActiveX controls [Denning 1997], while not as trustworthy as other applications, are only authenticated with digital signatures and currently must be trusted with the full rights of the user.

Using *restricted contexts*, Windows 2000 enables programs to run with limited authority and access only a subset of the resources available to a user. For example downloaded code can be limited to accessing the user interface but not the file system or network. With proper configuration, a user can implement the policy of *least privilege* [Saltzer and Schroeder 1975], in which programs are only granted access to the resources necessary for their execution.

This paper presents the access control mechanisms in Windows 2000, with an explanation of the tradeoffs that were made in their design. As background, in Section 2 we describe the access control mechanisms in Windows NT, and the design of the Active Directory in Windows 2000. We then explain why new access control mechanisms are needed. This is followed by a description of the extensions to support large numbers of rights for an object in Section 3 and improvements for centralizing management of access control in Section 4. In Section 5 we describe changes to limit the rights of untrusted code. In Section 6, we discuss previous work on these problems and then conclude in Section 7.

2. BACKGROUND

To explain the access control extensions in Windows 2000, we first describe access control in Windows NT 4.0. The access control mechanisms in Windows 2000 are an evolutionary step from the structures and mechanisms in Windows NT, both to maintain compatibility with existing applications and to minimize the changes to the operating system code. We will then describe the Active Directory and its security requirements.

2.1 Access Control Mechanisms in Windows NT 4.0

The security mechanisms of both Windows NT and Windows 2000 are built around *discretionary access control*, in which the owner of an object may specify who may access the object. These operating systems do not support *mandatory access control*, in which the system imposes a policy on all object accesses. Both operating systems distinguish *objects*, entities being accessed, from *subjects*, the active entities performing accesses. Access control in both operating systems is based on *access control lists*, which, for each object, specify the access granted to different subjects.

User SID	Jane User
Group SIDs	Administrators Service Operators Users
Privileges	Load Device Driver Shutdown System Take Ownership

Figure 1

ACL Header:	
Revision:	version 1
ACL Size:	100 bytes
ACE Count:	2
ACE 1:	
Type:	ACCESS_ALLOWED_ACE
Flags:	OBJECT_INHERIT
Access Rights:	read, write, delete
Principal SID:	Administrators
ACE 2:	
Type:	ACCESS_ALLOWED_ACE
Flags:	CONTAINER_INHERIT
Access Rights:	read, write, delete child
Principal SID:	Everyone

Figure 2

Figure 1 shows an example of a simplified access token in Windows NT, containing user and group identities and privileges. Figure 2 shows an example access control list (ACL) on a directory with two entries (ACEs).

2.1.1 Subjects A subject in Windows NT is represented by an access token, which is a kernel data structure storing a user's identity, group memberships, and privileges. Users may be organized into groups, such as all users that work on a project together. Users and groups are both *security principals*, which are the entities that may be granted or denied access to an object. Security identifiers, or SIDs, are variable-length byte strings that represent security principals. The operating system also supports a small number of standard privileges, which are represented as 64-bit numbers and have two purposes. First, privileges grant administrative access to a large set of objects, such as all files for backup/restore of disks or all drivers for system management. Second, privileges secure operations that have no specific object, such as shutting down the system or changing the system clock. Privileges may be granted to either users or groups, and the set of privileges granted to a security principal is stored in a policy database. Access tokens, shown in Figure 1, are constructed during logon by a trusted authentication service, which is responsible for authenticating the user and determining which user identifier, group identifiers, and privileges should be in the access token.

Every process in the operating system has an access token, and threads in a process may have a separate access token. If a thread has an access token, then that token is used for access control on operations made by the thread. If a thread has no access token, then the token from its owning process is used instead. Each process is either explicitly assigned an access token during creation (which requires the *SeAssignPrimaryToken* privilege) or copies the token from its creator. After a process starts, its access token may not be replaced, and the only change possible to its access token is to enable or disable privileges. For example, it is impossible to add or remove groups or change the user identity in an access token. Programs may replace the access token of a thread, though, which allows programs to *impersonate* users and perform specific actions with their identity. For example, a file server may assign the access token of its client to the thread processing the client's requests, so that calls to the file system to access the user's files will receive proper access control. For inter-process communication, the operating system kernel copies the access token between the client and server. For network communication, the authentication protocol is responsible for carrying client identities to the server, where a new access token is constructed. The token of a process that is impersonating is not used for access control, so a program run by a non-privileged user may enhance its access to a resource by impersonating the access token of a user who is granted more rights to the resource. As a result, administrators must take care not authenticate to programs run by non-privileged users.

2.1.2 Access Control. Windows NT provides a single major access control mechanism for all system resources that need access control, such as files, user interface objects, and kernel objects. Each object requiring protection is assigned a *security descriptor*, which stores all of its security state: the owner, group (used for emulating the Unix owning group field), access control list (ACL), and auditing information. The operating system provides a standard representation of access control lists for use by both system services and applications, removing the need for each application to implement its own access control algorithms and structures.

An access control list is a container for *access control entries* (ACEs), which determine which access rights should be granted or denied to specific security principals. An ACL may contain an arbitrary number of ACEs for different users or groups of users. The two types of ACEs in Windows NT are ACCESS_ALLOWED_ACEs, which grant a principal access, and ACCESS_DENIED_ACEs, which deny access. A sample ACL with two access control entries is shown in Figure 2. The ACE type field controls whether the ACE grants or denies access and the flags control whether it is copied onto the ACL of new objects. The remainder of the ACE contains the security identifier of the principal it grants or denies access and an *access mask*, which is a bit-field specifying the access rights. Windows NT allows just sixteen bits to be defined by the implementor of an object for specific access rights. In addition to controlling access to objects, rights in access control entries on containers may apply to the objects within the container. For example, the access rights on directories in a file system are: list directory, add file, add subdirectory, delete child, traverse, read attributes, and write attributes.

```
BOOLEAN
AccessCheck(Acl: ACL,
           DesiredAccess : AccessMask,
           PrincipalSids : SET of Sid)
VAR
  Denied : AccessMask = ∅;
  Granted : AccessMask = ∅;
  Ace : ACE;
foreach Ace ∈ Acl
  if Ace.SID ∈ PrincipalSids
    if Ace.type = GRANT
      Granted = Granted ∪ (Ace.AccessMask - Denied);
    else if Ace.type = DENY
      Denied = Denied ∪ (Ace.AccessMaska - Granted);
    if DesiredAccess ⊆ Granted
      return SUCCESS;

return FAILURE;
```

Figure 3: The algorithm for determining access control in Windows NT 4.0. The routine is simplified for clarification.

Windows NT places the responsibility for performing access control for both system components and applications on the *security reference monitor* in the operating system's kernel. The reference monitor concept, first described in [Anderson 1972], ensures that the access control algorithms are applied uniformly for all applications and system services. Applications protecting their own objects typically call the *AccessCheck* routine when an object is first accessed or opened. Subsequent accesses to the object that require only rights granted by a previous call to *AccessCheck* need not be checked (which may cause time-of-check to time-of-use errors). The *AccessCheck* routine passes the ACL, requested access rights, and the subject's access token into the security reference monitor. The entries from the ACL are evaluated in order, and each entry's SID is compared against the user and group SIDs in the subject's access token. If the SID is found, then the access rights in the ACE are either granted or denied, according to the type of ACE. Once a right has been denied, it may not be granted later by another ACE. Similarly, once a right has been granted, it may not be denied later. This algorithm is shown in Figure 3.

Interleaving allow and deny ACEs enables Windows NT to emulate Unix file system ACLs [Ritchie and Thompson 1974], in which only one entry in the list is ever used to grant a user access. For example, an ACL containing an entry granting a group access to a file followed by an entry denying the group all other accesses ensures that a member of that group receives the granted accesses and no more. The second ACE denies all other accesses and halts further evaluation of the ACL. While the security reference monitor supports any ordering of ACEs, the convention in Windows NT 4.0 is to place ACEs denying access before ACEs granting access, so that deny entries take precedence[1].

Table 1: Flags in the access control entry that control inheritance.

INHERIT_ONLY_ACE	ACE is only used for inheritance; it is not applied to this object
NO_PROPAGATE_INHERIT	ACE is inherited onto sub-objects, but no further
OBJECT_INHERIT_ACE	ACE is inherited onto sub-objects
CONTAINER_INHERIT_ACE	ACE is inherited onto sub-containers

2.1.3 Assigning Access Control While there are routines to directly modify existing access control lists and an ACL may be specified during object creation, in Windows NT they are commonly created by copying entries from the ACL on the container of an object. For example, when a file is created in a directory, access control entries from the directory's ACL are *inherited* and copied onto the ACL of the new file. Inheritance flags in each ACE, shown in Table 1, select which entries are copied onto the ACL, and these flags distinguish between entries that are copied onto files and onto directories, because they support different operations, such as reading data versus listing files. The OBJECT_INHERIT_ACE flag causes an entry to be copied from directories onto entities that are not containers, e.g. files, and the CONTAINER_INHERIT_ACE causes an entry to be copied onto entities that may contain other objects, e.g. directories. The INHERIT_ONLY_ACE marks entries on a directory that are not used for granting access to the directory but are instead intended only for inheritance. This flag is removed when the ACE is inherited. Finally, the NO_PROPAGATE_INHERIT flag limits the inheritance by removing all inheritance flags after the ACE has been copied. As a result, an entry marked NO_PROPAGATE_INHERIT and CONTAINER_INHERIT_ACE will be copied into the ACL of a new directory but not to any directories below it. This algorithm is shown in Figure 4.

[1] The Windows NT 4.0 ACL editor does not support deny entries at all (the tool does not correctly display ACLs with deny entries), which simplifies the user interface.

```
ACL
Inherit(ParentAcl: ACL, IsContainer : BOOLEAN)
VAR
  Ace : ACE;
  ChileAce : ACE;
  ChildAcl : ACL = ∅;
foreach Ace ∈ ParentAcl
  ChildAce = ∅;
  if IsContainer
    if container_inherit ∈ Ace.Flags
      ChildAce = Ace;
      ChildAce.Flags -= inherit_only;
    else if object_inherit ∈ Ace.Flags
      ChildAce = ACE;
      ChildAce.Flags += inherit_only;
  else
    if object_inherit ∈ Ace.Flags
      ChildAce = Ace;
      ChildAce.Flags = 0;
  ChildAcl += ChildAce;

return ChildAcl;
```

Figure 4: Algorithm for inheriting ACEs from a parent container onto a child object or container. The algorithm is simplified and does not include processing the NO_PROPAGATE_INHERIT flag, the CREATOR_OWNER identity, or generic rights.

Windows NT 4.0 includes two special mechanisms to allow slight variation in ACLs between different objects within the container. First, an inheritable ACE may contain the special identity CREATOR_OWNER. When an entry with this SID is inherited, the principal in the resulting ACE is replaced with the creator of the object. As a result, an ACE on a container can specify that the creators of objects within the container be granted specific rights. Second, inheritable ACEs may contain special access rights, called generic rights that are replaced during inheritance with rights specific to the object being created. For example, the ACL on a directory could contain an ACE granting generic read access. When this entry is inherited, the generic read access right is replaced with a set of rights, specific to the type of object created (either a file or a directory in this case), corresponding to read access. This mechanism allows read access to be interpreted as a set of finer-grained access rights, such as “read data” and “read attributes” for a file.

ACL inheritance allows permissions set on a directory to be propagated to every new file and directory created within it, but has no impact on already existing files and directories. System management tools simulate ACL inheritance onto existing files by reapplying inheritance on all the objects under a directory. However, if the user changing access control on a directory does not have permission to modify the ACL on an object lower in the tree, then inheritance is not applied to that object. Furthermore, there are no rules specifying how to merge existing ACLs with changes inherited from a parent, so the

Windows NT ACL editor discards the ACL on an object and create a new ACL consisting only of inherited entries.

Two separate mechanisms in Windows NT grant users permission to modify ACLs. First, access may be explicitly granted with the WRITE_DAC (write discretionary access control) access right. This right grants the permission to add or remove ACEs from the ACL. Second, the owner of an object is automatically granted the right to read and modify the access control list of the object, and can therefore always manage the object's ACL. The owner is initially set to the creator of an object, but it may be reset by use of the *SeTakeOwnershipPrivilege* privilege. This privilege allows a user to change the owner field of a security descriptor to their own identity, after which she may change the ACL. Because this privilege is normally granted to administrators, it is impossible to prevent administrators from accessing an object.

2.2 Active Directory

Despite the rich support for access control in Windows NT, the Active Directory requires additional capabilities. The Active Directory is the central point for system and application management in Windows 2000 and stores many types of information [Isenger 2000]. The data in the Active Directory are arranged as a hierarchy (single rooted tree) of typed objects, and each type has a common set of data properties and behaviors. Many objects are used for multiple applications. For example, user account objects are used for authentication by the Kerberos protocol as well as by the Exchange mail server for email delivery. As a result, a single object must be manageable by multiple sets of administrators. There are properties that are common to many types of objects, in particular metadata about the object, such as its type and its name, as well as properties that are unique to a single type. Finally, the Active Directory may be dynamically extended with new object types and by adding new data properties to existing object types. For example, a web publishing application could store the default web page for each user as a new property on user account objects.

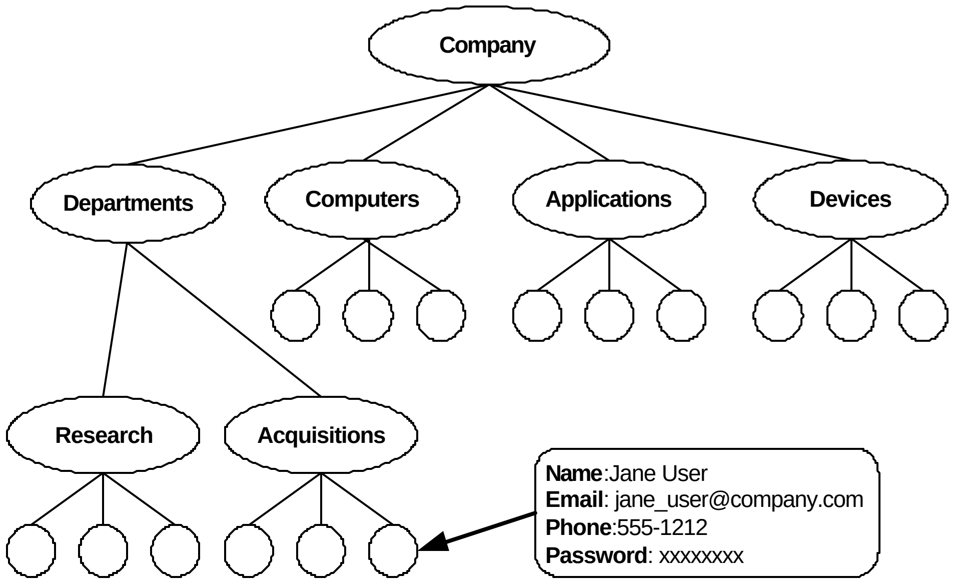


Figure 5: An example hierarchy in the Active Directory, showing multiple levels of containers with objects. Each object has a set of properties, as demonstrated by the properties on the object under the Marketing container.

The hierarchical structure of the Active Directory serves several purposes. First, it groups objects with similar management properties, such user account objects of people in one department of a company. Containers also group objects with similar types, such as applications' configuration data. Despite presenting data as a hierarchy, the Active Directory internally stores data in a flat database and maintains indexes over the full name of each object as well as other important properties. Figure 5 shows a sample directory layout with many containers, each with many objects. Each object has a collection of properties, such as a name and password for a user.

Table 2 presents definitions of terms relating to the Active Directory and access control in Windows NT and 2000.

Table 2: Definition of terms concerning access control and the Active Directory.

Term	Definition
ACE (access control entry)	Entry in an ACL that specifies, for a security principal, the access granted or denied to an object.
ACL (access control list)	List specifying, for an object, access granted or denied to a subject, containing access control entries (ACEs).
Access Token	Kernel data structure containing identity and privilege information about users active on a system.
Container	An application or system element that may have other elements with ACLs below it or contained within it, such as a directory in the file system.
Delegation	Granting another subject the permission to perform acts on one's behalf, either by modifying access control lists or modifying the subject of access control.
GUID (globally unique identifier)	A 128-bit number used for identifying object types in the Active Directory.
Impersonation	Performing an action with the identity of another subject.
Inheritance	The process of selecting access control entries on the ACL of a container to copy onto the ACLs of objects within the container.
Object	An application or system element that may be secured with an ACL and that does not have any elements with ACLs below it or contained within it, such as a file in the file system. Also a generic term describing any resource protected by an ACL, including containers.
Object Type	The description of the properties and behavior of a group of objects in the Active Directory, such as all objects representing users.
Privilege	System-wide right for a user to perform an operation, such as load a driver or backup a file.
Property	A name and value pair on an object in the Active Directory.
Property set	A group of name and value pairs with common access control policy in the Active Directory.
Security Descriptor	Container for security information about an object, including ACL, auditing information, and object's owner.
Security Principal	A user or group, identified by a security identifier (SID).
SID (security identifier)	Variable length byte string identifying users and groups.
Subject	An entity that performs operations, such as a program running on behalf of a user.

2.3 Limitations of Windows NT Access Control

The Windows NT access control structures and mechanisms are powerful and flexible, and can emulate other forms of access control lists, such as Unix file system ACLs [Ritchie and Thompson 1974] and DCE ACLs [Mackey and Salz 1993]. They may also be used to secure applications, such as databases or web servers. However, access control lists in Windows NT are optimized for applications with only a small number of types that are not extensible, such as files and directories in a file system. The mechanisms have several limitations with respect to Windows 2000 and the Active Directory:

1. Access masks are only sixteen bits, so a single ACL can only control sixteen different access rights.
2. Inheritance does not distinguish between objects with different access rights, and ACLs cannot be propagated to a tree of objects if some of the objects have ACLs that are not inherited.

3. There is no mechanism for restricting the rights of a program other than disabling privileges.

The first two flaws arise in the context of the Active Directory because of its many object types and many properties on each object type. Managing the Active Directory requires that access control must be administrable from the top of the directory hierarchy, so that an administrator may delegate control by granting other administrators access to all instances of a type of object.

The third flaw was exposed by the growing number of Windows NT systems connected to the Internet, which resulted in security exploits of network applications such as web browsers and email clients. As a partial solution, programs must be prevented from unnecessary access to user and system resources. This flaw, as well as the difficulty of supporting the Active Directory, forced us to update the access control mechanisms for Windows 2000.

2.4 Goals for Windows 2000 Access Control

Our goal as designers of access control in Windows 2000 was primarily to rectify the limitations of Windows NT 4.0. We wanted to allow ACLs to control access over an arbitrary, extendable, number of rights, so that a single ACL could protect an entry in the Active Directory that has many properties. We also wanted to allow administrators to set access control at a single point in the Active Directory, and let that policy flow to all appropriate objects below that point. Finally, we wanted to allow users to be able to safely download programs from the Internet and execute them, knowing that the programs could not damage their system or misuse their data. In the following sections we will first present our solution for extending the number of rights in an access control entry, followed by improvements to the access control inheritance mechanisms, and finally our mechanism for restricting the rights of a program.

3. TYPE-SPECIFIC ACCESS CONTROL

The access control entries in Windows NT 4.0 are unable to protect objects in the Active Directory because access masks are limited to only sixteen separate access rights. The directory service requires an access right to create each type of object and to access each property on an object, so the set of sixteen rights limits both the number of object types and the number of properties with different access control on an object. In addition, the Active Directory supports adding both new object types and new properties to existing objects, so the set of access rights for an object may be dynamically extended. For example, it would be impossible to add a new property to a user object with unique access control needs once the sixteen available rights have been used for other properties.

We considered storing a separate ACL for each property on an object. However, the existing ACL data structure is a simple container, so there is no need to duplicate the ACL itself. In addition, existing routines for managing ACLs in Windows NT are not equipped to manage multiple ACLs on a single object. It is also difficult to share access control entries between different properties when each property has a separate ACL. Another possibility we considered was to extend the access mask format so that more than sixteen bits are used to represent rights. However, we believe it is difficult to manage a bit field when properties are added or removed from an object. The solution we chose for Windows 2000 was to create a new access control entry format with a field that specifies the property on the object, or, in the case of creating and deleting child objects in a container, the type of object to which the ACE applies. To reduce the cost of protecting objects with many properties, we allow groups of properties to be protected with a single entry. This combination allows a small access control list to protect objects with simple needs while still allowing the full flexibility of protecting every property separately.

3.1 Object Types in ACEs

The new ACE format introduced in Windows 2000 adds two fields to each entry. The first new field, named *ObjectType*, identifies the scope of the access control entry. For directory service entries, the field identifies either the property or, for access rights on containers, the type of child object to which the entry applies. Other applications may use the fields for other purposes. The second new field, *InheritedObjectType*, controls which types of objects inherit the ACE and will be discussed in Section 4. Both fields are represented as GUIDs [Leach and Salz 1998], which are sixteen-byte values used by DCOM [Eddon and Eddon 1998] and the Active Directory to identify object types. An example of the new ACCESS_ALLOWED_OBJECT_ACE structure is shown in Figure 6.

ACE :	
Type:	ACCESS_ALLOWED_OBJECT_ACE
Flags:	OBJECT_INHERIT
ObjectType:	**LoginScriptPath**
InheritedObjectType:	**Users**
Access Rights:	read, write
Principal SID:	Administrators

Figure 6: Example object type specific ACE granting administrators access to the logon script path. This ACE is inherited onto user account objects. The new fields are shown in boldface.

The *ObjectType* field extends the set of rights available for an object by creating many sets of rights, each with a different GUID. Applications must supply an object-type GUID to the new *AccessCheckByType* routine to select the ACEs to evaluate. Only ACEs with a matching object type and those with no object type are evaluated. In addition, applications may extend the set of rights available at any time by creating new object-type GUIDs for use in access control entries. Object-type GUIDs have two uses within the Active Directory. First, every property on every type of object is assigned a GUID. The Active Directory protects properties with ACEs specifying the GUID of the property and the granted access, read or write. Second, every type of object is assigned a GUID, which is used for the right to create and delete objects in a container. The Active Directory grants the right to create a specific type of object within a container with an ACE specifying the desired object type and the create child access right. Because the same rights apply to all properties and all object types, ACEs with no object-type GUID can be interpreted as applying to either all properties, in the case of read and write, or all object types, in the case of create and delete child. These semantics are particular to the Active Directory, which has very regular objects, and other applications may choose to not use ACEs without object types, but for the Active Directory it is very concise to grant an administrator access to every property on an object or to create any type of object.

3.2 Property Sets

Specifying individual properties in ACEs provides fine-grained access control at the cost of greatly increasing the number of entries in an ACL. In Windows NT, a single access right often grants access to multiple operations or data properties, such as all the attributes on a file. In addition, a single ACE could grant access to any subset of the available access rights. With object-type specific ACEs, though, a separate entry is needed for each distinct property, which greatly increases the number of ACEs required. The number of calls to check access also increases when multiple properties are accessed, because each property must be checked separately. To counter both of these potential costs, we introduced a new access check routine, *AccessCheckByTypeResultList*, which checks for access to multiple properties in a single operation and returns a separate result for each property. In addition, this routine allows properties to be grouped into property sets, so that ACLs only need a single ACE to grant access to all properties in the set. Property sets are identified by a GUID, and access to a property is granted if the access is granted to its property set.

```
BOOLEAN
AccessCheckByType(Acl : ACL,
                  DesiredAccess : AccessMask,
                  PrincipalSids : Set of SID,
                  ObjectGuids : Set of GUID)
VAR
  Denied : AccessMask = ∅;
  Granted : AccessMask = ∅;
  Ace : ACE;
foreach Ace ∈ Acl
  if Ace.SID ∈ PrincipalSids &&
      (Ace.ObjectGuid == NULL || Ace.ObjectGuid Î ObjectGuids)
    if Ace.type = GRANT
      Granted = Granted ∪ (Ace.AccessMask - Denied);
    else if Ace.type = DENY
      Denied = Denied ∪ (Ace.AccessMask - Granted);
    if DesiredAccess ⊆ Granted
      return SUCCESS;
return FAILURE;
```

Figure 7: The algorithm for checking access to a single property with object-type GUIDs. The changes from Windows NT 4.0 are shown in italics. The GUIDs for the property and property set are represented as a set for simplicity.

Property sets are not visible within the structure of an access control entry; ACEs do not specify whether the object-type GUID refers to an object type (in the case of object creation), a property set or a property. Instead, the hierarchy is passed into *AccessCheckByTypeResultList*. This routine takes a list of property GUIDs and their containing property set GUIDs. The list must be in depth-first order, with each property set followed by the desired properties within it. The hierarchy allows ACEs granting and denying access to be correctly interpreted, so that a separate access check result can be returned for each property requested. The code for checking access with a single property is shown in Figure 7. The data structure supplying the list of properties and property sets forces properties to only belong to a single property set, because properties must follow their owning property set in the list of GUIDs.

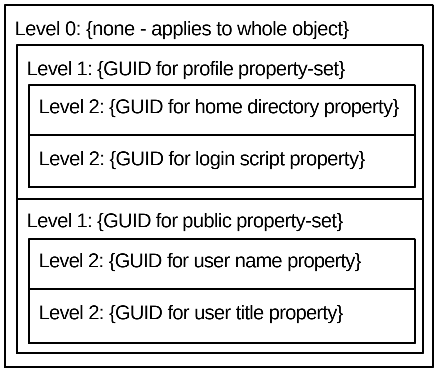
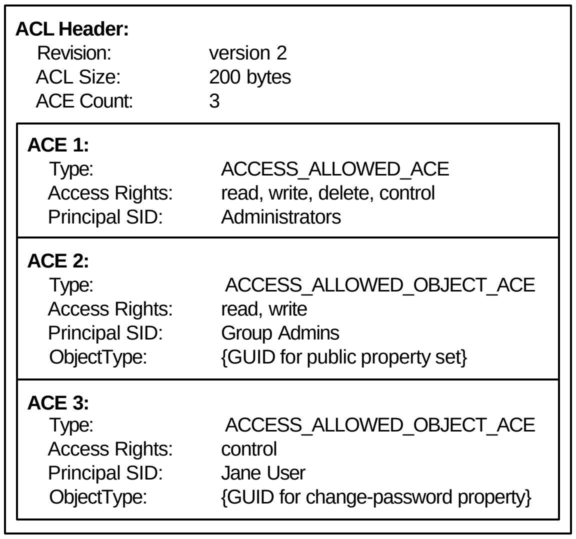

Level 0: {none - applies to whole object}
Level 1: {GUID for profile property-set}
Level 2: {GUID for home directory property}
Level 2: {GUID for login script property}
Level 1: {GUID for public property-set}
Level 2: {GUID for user name property}
Level 2: {GUID for user title property}

Figure 8: A multi-level list of object-type GUIDs used for an access check. The level indicates the scope of the GUID, either the whole object, a property set, or a property. Access to a property is controlled by ACEs with the GUIDs for its property set or with no GUID.

An example of such a list is shown in Figure 8. The list specifies which ACEs should be evaluated, and each level specifies that an ACE granting access at that level also grants access to GUIDs following it at a higher level. Level zero has no GUID, meaning that ACEs without GUIDs should be interpreted as applying to all properties and property sets. Similarly, ACEs granting access to property set GUIDs at level one also grant access to all the properties at level two in that property set. While the Active Directory only uses two levels, properties and property sets, the *AccessCheckByTypeResultList* routine supports up to five levels of nesting. Property sets enable compact ACLs for the common case when only a few different types of access are needed while allowing the complete flexibility of specifying access control separately on each property.



ACL Header:

Revision:	version 2
ACL Size:	200 bytes
ACE Count:	3

ACE 1:

Type:	ACCESS_ALLOWED_ACE
Access Rights:	read, write, delete, control
Principal SID:	Administrators

ACE 2:

Type:	ACCESS_ALLOWED_OBJECT_ACE
Access Rights:	read, write
Principal SID:	Group Admins
ObjectType:	{GUID for public property set}

ACE 3:

Type:	ACCESS_ALLOWED_OBJECT_ACE
Access Rights:	control
Principal SID:	Jane User
ObjectType:	{GUID for change-password property}

Figure 9: A sample ACL for a user object using object-type specific access control entries.

3.3 Example

Figure 9 demonstrates how object types and ACEs are used and shows an ACL on a user object in the Active Directory. The first ACE grants administrators full control over all the properties of the user. The second ACE grants group administrators read and write access to the user's public information, such as her phone number. The third ACE grants the user herself access to change the password on the account. For access rights that correspond to executing a procedure and not accessing data, the Active Directory defines the right *control*. In this case, the Active Directory understands that a password change protocol is allowed to change a user's password if it can prove knowledge of the user's previous password. This ACL demonstrates the space saving of property sets, because many additional entries would be required to specify access separately for each property in the 'public' property set.

3.4 Discussion

The primary purpose of type-specific ACEs is to allow applications to both have a large set of access rights as well as to dynamically extend their set of rights. Hierarchically grouping access rights into property sets simplifies management by allowing administrators to grant access to a single property set instead of separately granting access to all the properties within it. In addition, property sets simplify extending types, because no changes to access control lists are needed when a property is added to or removed from an existing property set. Property sets also lessen the memory and performance impact of a large set of similar rights by allowing many rights to be coalesced into a single access control entry. Finally, property sets simplify the user interface by allowing the display of a smaller number of property sets rather than (potentially) hundreds of individual p roperties.

We found that properties and property sets simplify debugging of access control problems, because there is a clear mapping between access control entries and rights to an object. Previously, in Windows NT a single access right frequently controlled access to many properties, and there was no mechanism to determine which right controlled which properties. As a result, the user interface tools were hard-coded with names for access rights instead of the list of properties actually protected. By formalizing the mapping, object-type specific access control entries specify exactly which properties may be accessed. The tools also no longer rely on a fixed mapping between rights and properties, because they may query the directory service for the names of properties and the members of property sets listed in an ACL.

Despite the fine-grained control offered by per-property access control, there has been some customer resistance to specifying access rights on individual properties. In particular, administrators find it difficult to manage a large number of properties individually, so access control changes are usually applied at the property set level. It is not yet clear whether the fine granularity of protecting individual properties is necessary if property sets are well chosen.

If there is an inexact match between property sets and administrative needs, then the property set mechanism breaks down and ACLs can become bloated with entries for individual properties. For example, this can occur when upgrading Windows NT 4.0 servers to Windows 2000. The access control lists protecting user account objects in the Windows NT 4.0 directory service are converted into ACLs with type-specific ACEs. Each access right for a user object in Windows NT 4.0 grants access to many properties, and these groups of properties do not map perfectly onto property sets in Windows 2000. Rather than converting an access right in Windows NT 4.0 into an access right to a Windows 2000 property set, the upgrade process converts it instead into a sequence of ACEs granting access to each property. The resulting access control lists can be many times longer than those on user account objects created natively in Windows 2000. For example, the ACL on a user account upgraded from Windows NT 4.0 may contain more than 30 ACEs granting access to specific properties.

There has been some feedback from developers indicating that allowing a property to be a member of multiple property sets would simplify administration and shrink the size of access control lists. For example, some properties may logically belong to multiple property sets, such as an email address field on a user that may be part of both a contact information property set and an email property set. This change, though, would make it difficult to understand when access is granted, and is probably better solved through property-specific ACEs.

The primary drawback of storing object-type information in access control entries is the increased cost of both storing access control lists and performing access checks when there are many properties on an object and access to many properties is being requested. For example, it is not uncommon for ACLs in the Active Directory to be more than 1600 bytes, and for users to request access to ten or more properties. In Windows 2000, the NTFS file system reduces the storage cost by only storing one instance of each unique ACL. Objects in the file system reference the specific ACL rather than storing a separate copy. In the successor to Windows 2000 Server we added these optimizations to the Active Directory. Another potential improvement for evaluating ACLs is to cluster the ACEs in an ACL that grant access to a particular property to reduce the number of entries that must be inspected.

Despite these drawbacks, type-specific access control is crucial to the Active Directory. This extension allows the Active Directory to use common operating system functionality, so that a single permissions editor may be used for both it and the rest of the operating system. In addition, it shares the reference monitor and security infrastructure, which minimizes the amount of new code that must be trusted. The additional features allow administrators to grant access to only the properties needed for a job function or application instead of all those controlled by a more general access right. At the same time, property sets preserve fine-grained control while optimizing for the common case where many properties share the same access control. Finally, identifying properties by GUIDs simplifies adding and removing properties from an object by leaving ACEs for other properties unchanged.

4. INHERITANCE CONTROL

Windows NT assigns access control to new objects primarily through inheritance of access control entries from the ACLs on containers. There are two major limitations to the inheritance mechanisms in Windows NT:

1. It is impossible to specify that different access control lists be inherited onto different types of objects within a container.
2. It is difficult to propagate changes to ACLs through a tree of objects, because inheritance rules cannot be reapplied without erasing any modified ACLs lower in the tree.

Both problems arise in the Active Directory, because of its many types of objects and the many different administrators of these objects. One of the goals for the Active Directory is to allow *delegation* of administration, so that one user can grant another user control over a subset of the objects in the directory service, such as all printers or objects for a department. Administrators must therefore be able to change permissions at one place in the directory and let the effects propagate down either to all objects or only those of the appropriate type. Using the access control inheritance mechanism from Windows NT, which was designed with a file system in mind, all objects within a container inherit the same access control. Furthermore, changes to access control at the root of a tree overwrite all changes lower in the tree. Consider a directory service with user and printer objects and a separate container for each department in a company. Using inheritance to grant a printer operator access to all the printers in one department requires that the administrator also grant access to all the user account objects, which is unnecessary. Thus, changes to the ACL inheritance mechanism were needed to support the Active Directory. The file system also benefits from inheritance changes, because changes to ACLs are common there as well.

We considered several solutions for each problem. One solution for supporting multiple object types, similar to the design of ACLs in DCE [Mackey and Salz 1993], is to store multiple ACLs on a container, one for each object type. However, as with storing an ACL for each property, that approach may be inefficient when many ACEs are common to all child objects because the entries must be duplicated in each ACL. As discussed in Section 3, the routines for manipulating access control data in Windows NT do not support multiple ACLs on a single object, so significant changes would be needed for managing multiple ACLs. Storing ACLs for the various types of objects in a separate database is another option. However, this solution alone does not allow hierarchical propagation of access rights. Instead, we chose to let applications annotate each ACE with the type of object that should inherit the ACE. When an object is created, only those ACEs with its type or no type are inherited onto the new object. A single ACL can then propagate different ACLs onto each type of object created below it.

To allow the granting of rights to a tree of objects, we considered *dynamic inheritance*: if an access right is not granted on an object, then access is checked on all parent containers until the right is granted or the root is reached. This approach is taken by the NDS directory service [Cadjan and Harris 1999]. However, we believe that the access control mechanism should not assume that because objects are named in a hierarchy they are also stored and accessed in a hierarchy. For example, files in NFS are accessed by file identifier, not by path [Callaghan et al. 1995]. Similarly, the Active Directory stores data in a flat database with an index over the full name of each object, so there is no convenient opportunity to access all the ancestors of an object. Furthermore, reading and writing an object is a common operation, while changes to ACLs are infrequent, so the performance of propagating ACL changes is not critical relative to the speed of an access check. Taking these conditions into account, our implementation uses *static inheritance*, in which inheritance is reapplied only when ACLs change and a new ACL is written to each object. Only a single ACL must be evaluated for most access checks. To propagate changes correctly, we annotate ACEs with a flag indicating whether they were inherited so that the locally applied ACEs can be identified and preserved when inheritance is reapplied. In addition, the inheritance mechanism for ACLs is idempotent, so that it can be re-applied after a failure.

4.1 TYPE-SPECIFIC INHERITANCE

Similar to type-specific access control, Windows 2000 allows type-specific inheritance. Applications with multiple types of objects mark ACEs with a new field, *InheritedObjectType*, which specifies the type of object that inherits the ACE. When an object is created, those ACEs without an inherited object type or with a matching inherited object type are copied into its ACL. Similarly, when an access control change is propagated, just the matching entries for each type of object are copied onto the objects in a container. Inheritance still distinguishes between containers and objects, because containers must be able to propagate access control to their children and therefore copy all inheritable ACEs. Objects, in contrast, only require ACEs that apply to the object itself. In Windows XP, the successor to Windows 2000, type-specific inheritance is extended to support multiple inheritance, so that an object may inherit the access control entries for multiple object types. Multiple inheritance allows new specialized types, such as “web-users”, which share the properties and inherit the access control of normal users in addition to ACEs only needed for web-users. Figure 10 shows pseudo-code for the inheritance algorithm.

```
ACL
Inherit(ParentAcl: ACL,
       ChildAcl: ACL,
        ChildType : GUID,
        IsContainer : BOOLEAN)
VAR
  Ace : ACE;
  ChileAce : ACE;

/* remove inherited ACEs from child ACL */
foreach Ace Î ChildAcl
  if inherited_ace Î Ace.flags
    ChildAcl -= Ace;

/* add inheritable ACEs from parent ACL */
foreach Ace in ParentAcl
  ChildAce = ∅;
  if IsContainer
    if container_inherit ∈ Ace.Flags
      ChildAce = Ace;
      ChildAce.Flags -= inherit_only;
      ChildAce.Flags += inherited_ace;
      /* Mark ACEs without a matching type as
         inherit only */
      if (Ace.InheritedObjectGuid != NULL) &&
         (Ace.InheritedObjectGuid != ChildType)
        ChildAce.flags |= inherit_only;
    else if object_inherit ∈ Ace.Flags
      ChildAce = ACE;
      ChildAce.Flags += inherit_only + inherited_ace;
  else if (object_inherit ∈ Ace.Flags ) &&
         ( (Ace.InheritedObjectGuid == NULL) ||
           (Ace.InheritedObjectGuid == ChildType) )
      ChildAce= ACE;
      ChildAce.Flags = inherited_ace;

  ChildAcl += ChildAce;

return ChildAcl;
```

Figure 10: Algorithm for type-specific inheritance. The changes from Windows NT 4.0 are shown in italics. The first change removes inherited ACEs, to ensure that inheritance is idempotent. The second change marks all inherited ACEs, so they can be removed in the future. The remaining changes verify that the *InheritedObjectGuid* matches either the type of the object receiving access control, or is empty. The algorithm is simplified and does not include processing of the NO_PROPAGATE_INHERIT flag.

In order to scale to a large number of objects, we take advantage of the fact that not all object types require a variety of different ACLs. The Active Directory implements a database of default ACLs that are placed on all objects when they are created. Only objects with ACLs that vary in different portions of the directory hierarchy require type-specific inheritance, which modifies the default ACL. Thus, for object types that only need a single ACL for each instance of the object, the default database supplies the ACL. For object types that have more varied access control, inheritance allows variation in ACLs. Thus, the combination of default ACLs and type-specific inheritance allows scalability to a large number of object types.

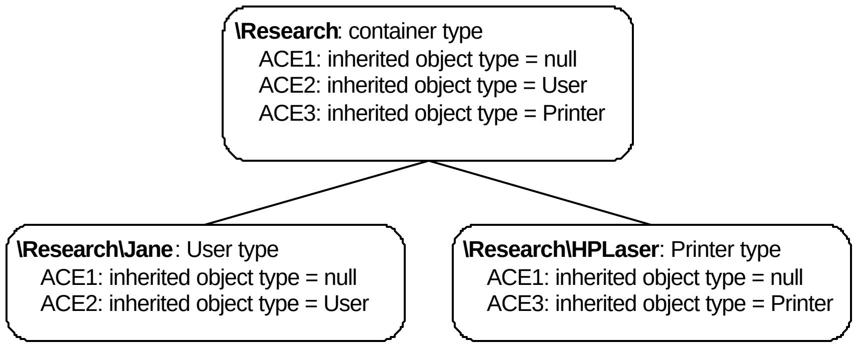

\Research: container type
ACE1: inherited object type = null
ACE2: inherited object type = User
ACE3: inherited object type = Printer
\Research\Jane: User type
ACE1: inherited object type = null
ACE2: inherited object type = User
\Research\HPLaser: Printer type
ACE1: inherited object type = null
ACE3: inherited object type = Printer

Figure 11: ACLs on a container and two objects within the container. The ACEs on the '\Research' container are inheritable onto different object types, so the two child objects receive a different set of ACEs.

Figure 11 shows an example of type-specific inheritance. In this example, the *Research* container has ACEs that are to be inherited onto all objects, *User* objects, and *Printer* objects. The user *Jane* inherits the ACEs with no inherited object type and with a User object type. Similarly, the *HPLaser* printer inherits the first ACE and the Printer ACE. This example demonstrates how a single ACL can inherit different ACEs onto different types of objects.

4.2 Static Inheritance

Dynamic inheritance, in which permissions for an object may be set on any container above the object, presents a simple and intuitive model of access control at the cost of checking access on many containers whenever an object is accessed. The Active Directory emulates dynamic inheritance by pre-computing the access control for an object when ACLs are changed rather than when access is requested. The difference is in implementation; the resulting permissions are the same. The primary difficulty in this illusion is merging ACEs applied locally to an ACL with the entries inherited from its parent. In addition, it must be possible to limit inheritance so that portions of a hierarchy, such as those containing private information, can override inheritance.

Windows 2000 enables modifications to access control lists to propagate down a tree by annotating ACEs with inheritance information. The algorithm for inheriting access control in Windows 2000 is idempotent, so that if the propagation of inheritance aborts due to a system failure, inheritance can be applied again with identical results. The inheritance mechanism is *static*, because inheritance is only evaluated when an ACL changes rather than during every access request. The resulting access for a principal is the same as if inheritance were dynamic and an object's ancestors were checked for access.

The ACL data structures in Windows 2000 annotate each access control entry with a flag indicating whether or not it was inherited. Each ACE that was inherited has the INHERITED_ACE flag set in its header. These ACEs are removed before reapplying inheritance, leaving only the entries added directly to the ACL. As a result, reapplying inheritance does not overwrite locally specified access control entries.

Inheritance onto an object or container may be disabled, to provide both local control and more restrictive access control for portions of the tree. For example, normal users are generally able to browse all the objects in the directory. However, an organization may want to exempt the acquisition department's objects, because the names of the objects may reveal privileged information. The SE_DACL_PROTECTED flag, which is stored on security descriptors, prevents any ACE from being inherited onto an ACL. The ability to set this flag, although not stored in an ACL, is granted by the same access right, WRITE_DAC, that controls the right to modify the ACL itself. An administrator may therefore create a more secure portion of the hierarchy by preventing inheritance of access rights.

In addition to adding these flags, the ordering rules for ACEs changed for Windows 2000. In Windows NT 4.0 it is recommended that ACEs denying access be placed first in an ACL, so that deny ACEs have precedence over allow ACEs. However, to follow the discretionary access control model, which allows the owner of an object control over who may access the object, we chose to grant administrators of a sub-tree the ability to override all inherited permissions, which results in interleaving grant and deny ACEs from each container on a single ACE. In addition, this rule provides a closer simulation of dynamic access control, in which access is checked by walking up the hierarchy of parent containers. The alternative of placing all ACEs denying access first prevents the administrator of an object from overriding an inherited ACE that denies access. Similarly, placing inherited ACEs first prevents the owner of an object from controlling the resulting access. Therefore, in Windows 2000, all locally added ACEs are placed first, followed by inherited entries. If the entries are inherited from containers at several levels in the tree, then the ACEs from closer containers will be ordered before ACEs from more distant containers. The administrator of an object retains full control over the ACL on the object, because she can either protect the object from inherited access control, or add explicit ACEs to the beginning of the ACL that are evaluated before (and hence override) inherited entries.

The implementation of inheritance in the Active Directory is significantly different than in the file system. In the file system, the management tools, such as the Windows Explorer, implement inheritance. When an ACL is changed, these tools walk the file system and write new ACLs on to every effected file or directory. If the user running the management tool does not have permission to modify the ACL on an object, then inheritance stops, even if the object does not explicitly block inheritance. Whether an ACL may be changed depends on the user running the management tool, which is confusing because it does not simulate the effect of dynamic inheritance. The Active Directory, however, propagates ACL changes itself rather than relying on a management tool. A user with permission to modify the ACL on a container implicitly has permission to modify the inheritable ACEs on all objects underneath that container, unless inheritance is explicitly blocked. As a result, changes to the ACLs at the root of a tree propagate completely, independent of the rights of the user who made the changes. It was our goal that the file system in Windows 2000 would also implement inheritance itself, but schedule pressures prevented that change.

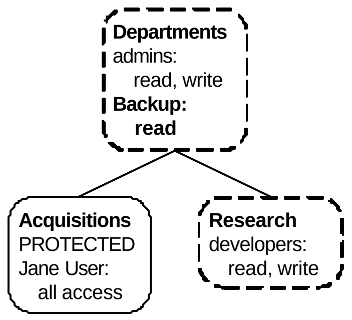


Figure 12: Dynamic inheritance. The entries on the *Departments* container are automatically inherited to the *Research* container during access. Adding a new entry requires updating a single ACL, as shown by the addition of a single ACE to grant Backup access. The ACLs referenced during an access check on the *Research* container are dashed.

4.3 Example

To demonstrate the desired effect of inheritance, Figure 12 shows an example of how dynamic inheritance can be applied to the directory service. In this example, the '*Acquisitions*' container overrides the inherited permissions by removing the access of administrators and instead grants access to the user '*Jane User*'. The other container, '*Research*', augments the inherited permissions by additionally granting the '*Developers*' group read access. When a new ACE is added to the '*Departments*' container, the change is effective for the *'Research'* container, while the *'Acquisitions'* container is protected from inheritance.

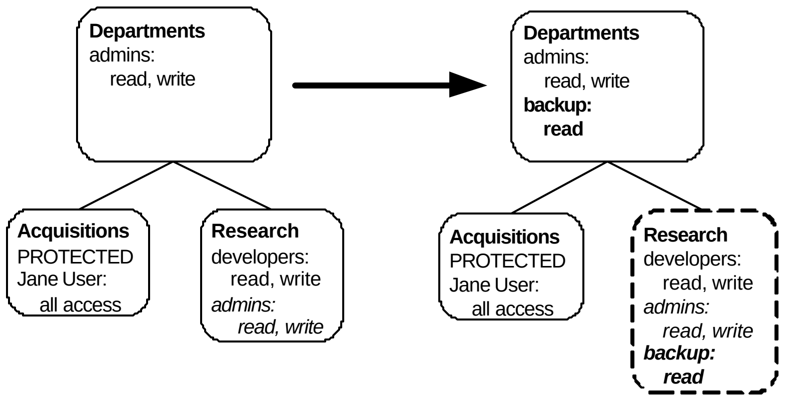


Figure 13: An example of reapplying inheritance. On the left are shown containers in a directory service, and on the right are the result ACLs after the bold-faced entry was added to the ACL on the *'Departments'* container. The ACLs referenced during an access check on the *Research* container are dashed.

Compared to dynamic inheritance, static inheritance results in changing more ACLs when access control changes, and also results in larger ACLs. Figure 13 shows the same example from above with static inheritance. The resulting access is the same, but the access control lists in this case are longer because information is duplicated on both the container and the child ACL. However, there benefit comes during an access check: with static inheritance, only the ACL on the object itself must be inspected. With dynamic inheritance, as shown in Figure 12, multiple ACLs must be inspected.

4.4 Semantics of Inheritance

Static inheritance allows complex access control policies to be expressed, such as specifying where certain types of objects may be created. To better specify exactly what policies may be expressed, we present a formal description of inheritance. Rules one and two below give the semantics of the OBJECT_INHERIT_FLAG in conjunction with object types. The first rule ensures that all objects with a matching type inherit the ACE, and that the ACE is used for access control (INHERIT_ONLY is turned off). The second rule ensures that all containers inherit all OBJECT_INHERIT ACEs as well, but do not use it for access control (INHERIT_ONLY is turned on).

$$\begin{aligned} &\forall containers\ C, \forall objects\ O, \forall ACEs\ A\ in\ ACL\ on\ C \mid \\ &\qquad A.OBJECT\_INHERIT = TRUE \wedge C \in ancestors(O) \\ &\qquad \wedge (O.type = A.inheritedObjectType \vee A.inheritedObjectType = NULL) \\ &\quad \Rightarrow O\ inherits\ A, A.INHERIT\_ONLY \leftarrow FALSE \end{aligned} \tag{1}$$

$$\begin{aligned}&\forall containers\ C, C', \forall ACEs\ A\ in\ ACL\ on\ C \mid \\ &\quad A.OBJECT\_INHERIT = TRUE \wedge C \in ancestors(C') \\ &\Rightarrow C'\ inherits\ A, A.INHERIT\_ONLY \leftarrow TRUE\end{aligned} \tag{2}$$

Similarly, the rules for the CONTAINER_INHERIT flag are given below. ACEs with CONTAINER_INHERIT are inherited to all containers and only those with a matching type use it for access control (rule three). On other containers, the ACE is marked as INHERIT_ONLY (rule four).

$$\begin{aligned}&\forall containers\ C, \forall containers\ C', \forall ACEs\ A\ in\ ACL\ on\ C \mid \\ &\quad A.CONTAINER\_INHERIT = TRUE \wedge C \in ancestors(C') \\ &\quad \wedge (C'.type = A.inheritedObjectType \vee A.inheritedObjectType = NULL) \\ &\Rightarrow C'\ inherits\ A, A.INHERIT\_ONLY \leftarrow FALSE\end{aligned} \tag{3}$$

$$\begin{aligned}&\forall containers\ C, \forall containers\ C', \forall ACEs\ A\ in\ ACL\ on\ C \mid \\ &\quad A.CONTAINER\_INHERIT = TRUE \wedge C \in ancestors(C') \\ &\quad \wedge (C'.type \neq A.inheritedObjectType \wedge A.inheritedObjectType \neq NULL) \\ &\Rightarrow C'\ inherits\ A, A.INHERIT\_ONLY \leftarrow TRUE\end{aligned} \tag{4}$$

The fifth and sixth rules order the ACEs in an ACL. By rule five, if two ACEs are ordered on a container and the ACEs are inherited to a child object or container, then the ACEs must be in the same order in both ACLs. By rule six, if one container is an ancestor of another, then the ancestor's ACEs will appear later in any ACL that inherits from both containers. This rule requires that containers be organized as a hierarchy, so that no container has more than one parent.

$$\begin{aligned}&\forall A, A'\ ACEs\ in\ ACL\ on\ container\ C, \forall objects, containers\ X \mid \\ &\quad C \in ancestors(X) \wedge A\ precedes\ A'\ in\ ACL\ on\ C \wedge X\ has\ A, A'\ in\ ACL \\ &\Rightarrow A\ precedes\ A'\ in\ ACL\ on\ X\end{aligned} \tag{5}$$

$$\begin{aligned}&\forall A, A'\ non-inherited\ ACEs, \forall objects, containers\ X, \exists\ containers\ C, C' \mid \\ &\quad A\ in\ ACL\ on\ container\ C \wedge A'\ in\ ACL\ on\ C' \\ &\quad \wedge C \in ancestors(C') \wedge C' \in ancestors(X) \wedge X\ has\ A, A'\ in\ ACL \\ &\Rightarrow A'\ precedes\ A\ in\ ACL\ on\ X\end{aligned} \tag{6}$$

The rules above ignore the possibility of a protected ACL. The seventh rule, shown below, limits the scope of the previous rules for objects and containers flagged with SE_DACL_PROTECTED. This rule ensures that none of the ACEs in a protected ACL are inherited.

$$\begin{aligned}&\forall object, container\ X, \forall ACEs\ A\ in\ ACL\ on\ X, \forall containers\ C \mid \\ &\quad X.SE\_DACL\_PROTECTED = TRUE \\ &\Rightarrow \neg(C \in ancestors(X) \wedge A\ in\ ACL\ on\ C)\end{aligned} \tag{7}$$

These rules ensure that access control decisions can be propagate fully through a tree of objects, stopped only by a protected ACL.

ACL Header:

Revision:	version 2
ACL Size:	400 bytes
ACE Count:	4

ACE 1:

Type:	ACCESS_ALLOWED_OBJECT_ACE
Access Rights:	write
Principal SID:	PRINCIPAL_SELF
Inherited Obj. Type:	{GUID for User Account Objects}
Object Type:	{GUID for WWW Homepage}

ACE 2:

Type:	ACCESS_ALLOWED_OBJECT_ACE
Access Rights:	create child
Principal SID:	Server Applcations
Inherited Obj. Type:	{GUID for RPC Services}
Object Type:	{GUID for RPC Endpoint}

ACE 3:

Type:	ACCESS_ALLOWED_OBJECT_ACE
Access Rights:	create child
Principal SID:	Administrators
Inherited Obj. Type:	{GUID for Organization Units}
Object Type:	{GUID for User Account Objects}

ACE 4:

Type:	ACCESS_DENY_OBJECT_ACE
Access Rights:	create child
Principal SID:	everyone
Inherited Obj. Type:	NULL
Object Type:	{GUID for User Account Objects}

Figure 14: Complex policies expressed with static and type-specific inheritance. The combination of the two mechanisms allows administrators to direct where certain objects can and cannot be created.

Figure 14 shows an ACL that takes advantage of both type-specific inheritance and static inheritance to express a complex policy. The first ACE depends on the first rule for inheriting onto objects and grants users the permission to set their own homepage for the World Wide Web. The PRINCIPAL_SELF SID in this ACE represents the user whose object is being protected, and is similar to the CREATOR_OWNER SID used for inheritance (Section 2.1.3). The service calling *AccessCheckByTypeResultList* can supply an arbitrary SID to replace the PRINCIPAL_SELF SID, unlike the CREATOR_OWNER SID, which is replaced during inheritance with an object's creator (because the creator's identity is stored with the object, in the security descriptor). The Active Directory passes in the SID of the object being protected, such as the SID of the user for user account objects and the SID of the group for group account objects. This mechanism allows an administrator to grant a user permission to modify portions of her own account object but not other users' account objects, and for group members to remove themselves from the group.

The second ACE depends on rule four for container inheritance, and allows server applications, such as a database or a web server, to create RPC endpoints in any container of type '*RPC Services*'. The key technique here is to create a container type for a specific type of object, and then use a type-specific ACE to grant access to create the object within the container. Finally, the third and fourth ACEs in Figure 14 restrict the type of container in which a user account object may be created. These ACEs depend on the fifth rule, which ensures that the order of the two ACEs is maintained whenever both ACEs are inherited. The third entry grants administrators the right to create users in *Organizational Units* and the final entry denies everyone the right to create a user in every type of container, due to the NULL inherited object type. Because these two ACEs are evaluated in order and inherited in order, a member of the administrators group will always be granted access to create a user in an organizational unit, but all other types of containers only the ACE denying permission to create users will be inherited, preventing everyone from creating users.

4.5 Discussion

Type-specific inheritance and static inheritance allow centralized management by propagating changes through a hierarchy of objects, so that access control changes are only made in one place. These features support delegation by allowing an administrator to grant access to a single type of object, or even a single property on a single type of object. In addition, that access is propagated to new objects when they are created. Portions of the tree may also be more protected and block inheritance of rights from above. This approach provides the major benefit of dynamic inheritance, which is centralized administration but lowers the cost at access time because ACLs along the whole path do not need to be evaluated.

In our experience, the new inheritance mechanisms are typically used for making global changes to the Active Directory, as they would to a file system, rather than to delegate access to particular object types. For example, administrators are currently wary of granting access to create printers and manage all printer objects by placing an access control entry at the root of the directory tree. Instead, they prefer to change the ACLs at each container with a printer. This again is a form of organizational resistance to distributing responsibility for objects in the directory and may change as more applications use it to store data. There has also been resistance by administrators to dividing the administration of an object between multiple individuals. For example, security administrators have been hesitant to allow email administrators the permission to modify any portion of a user object, even if the property relates only to email. However, this resistance may be an artifact from familiarity with Windows NT 4.0, in which each application stored its information separately. As more applications use the Active Directory, shared access to objects and split administration may become more common.

Similar to type-specific access control, static inheritance also simplifies both debugging and managing access control because the ACL on an object is usually the sole determinant of access to that object. Only a single ACL must be inspected to determine the access granted to a user instead of examining the ACLs from all the containers above an object. The user interface in Windows XP even displays the name of the object from which an ACE was inherited. In addition, compared to Windows NT, the explicit marking of inherited ACEs makes it easier to understand the source of each ACE in an ACL.

The user interaction with ACLs in Windows 2000, while benefiting from explicit marking of inherited ACEs, is complicated by the use of privileges and object ownership to grant access. The Novell Corporation, in a critique of the Active Directory [Novell Inc. 2000], complained of two issues. First, the privilege to take ownership of an object allows administrators to take complete control over all objects, because an administrator may change any object's ACL. The user interface, though, does not display this ability. Second, protecting ACLs from inheritance does not completely restrict access to those objects because the ability to take ownership overrides the protection provided by ACLs. As a result, it is impossible to prevent any portion of the Active Directory from being accessed by a user with the *SeTakeOwnershipPrivilege* privilege. However, as designers we agreed that it was important to grant some level of administrator access to the complete directory, to allow the organization to reclaim control of objects when a user departs or is unavailable [Microsoft Corp. 2000]. This approach, though, requires limiting the use of the take ownership privilege. Granting the right to manage access control through a privilege, though, complicates the user interface because the permissions editor displays only the contents of the ACL instead of the true permissions granted to the object.

Allowing owners to modify the ACL on an object also reduces the ability to restrict who can create certain object types. For example, organizations commonly want only security administrators to create user account objects. However, if a user can create any type of container, then that user, as owner of the container, may modify the ACL to grant herself the right to create new user account objects. The Active Directory avoids this limitation by storing, for each type of object, a list of container types that may contain the object. If creation of those containers is similarly controlled, then administrators can limit the creation of any type of object. User account objects, for example, may only be created within *organizational unit* containers, so limiting the right to create organization units also limits the right to create users. The need for these restrictions is due to the semantics of the directory service, where objects are accessed by queries rather than strictly by name. For example, a user account object may be used for authentication independent of where it is created. For applications where the location of an object is more significant, such as in a file system, the restriction on creating specific object types may not be needed. In addition, an application may implement a similar restriction without a separate list of valid container types by ensuring, with careful use of access control entries that unprivileged users are only allowed to create objects, and not containers.

The Active Directory further complicates static inheritance because the distinction between containers and objects is not fixed. Every object type has a property, *Container*, which indicates whether the object may be a container and have other objects below it. This flag may change value, so an object type that is initially not a container may later become a container. Rather than incur the complexity of updating ACLs when an object type is changed, the Active Directory instead treats all objects as containers, causing even larger ACLs.

Compared to Windows NT, this inheritance mechanism increases the cost of access control in space, time, and complexity. The inherited ACE information for each object type is duplicated in the ACL of every container, so it may require much more space to store. However, the Active Directory in the follow-on operating system to Windows 2000 Server stores ACLs in a shared table so that duplicate ACLs can be merged and only a single copy of each unique ACL is required. The use of a table of default ACLs greatly reduces the size of access control lists because not all object types require ACEs on containers.

The larger ACLs also make access check operations on containers more expensive [Microsoft Knowledge Base 2000], because every ACE, whether or not it impacts an access control decision, must be read and inspected. Although the speed of access checks has not been a problem in the Active Directory, caching the result of an access check can lower the cost of access control by not evaluating the same ACL multiple times.

Finally, applying inheritance statically requires that some piece of code walk the tree of objects and reapply inheritance. This reapplication must be resumed if the machine crashes, and for certain applications, such as the Active Directory, the reapplication must be transactional. The process is much more complex than dynamically applying inheritance during an access check, but is on a less frequent code path. Once implemented, static inheritance can provide the manageability benefits of dynamic inheritance with lower runtime costs.

5. PROTECTION FROM UNTRUSTED CODE

Fine-grained access control allows administrators to control which *users* may have access to an object, but it does not let users choose which *programs* may have access. The third major access control concern in designing Windows 2000 was preventing misbehaving programs from causing damage. One alternative, used by Tron [Berman et al. 1995] and Janus [Goldberg et al. 1996], is to augment the operating system with additional checks on the parameters of system calls. While such a mechanism could have worked in Windows 2000, the operating system already protects all its internal objects with ACLs and privileges. In addition, Windows 2000 has more than two hundred system calls, so trapping each one and verifying parameters separately is difficult. We believe that given the opportunity to modify the operating system, it is better to extend the existing operating system access control mechanisms rather than add a new set of mechanisms. It is easier to understand and administer a system with one set of mechanisms than one using different mechanisms for different access control purposes. Finally, multiple access control mechanisms protecting the same objects may cause confusing results or interact poorly, because it is difficult to predict program behavior.

Our solution, *restricted contexts*, is based on three goals:

1. Untrusted code should have no greater access to resources than the user running the code.
2. Users should be able to restrict programs to accessing specific objects or classes of objects.
3. No separate security model, beyond the operating system's protection and access control model, should be needed for restricting code.

These goals suggest that untrusted code should use operating systems protection mechanisms by running in a separate process and address space with its own access token, and access control on objects should limit the code to a subset of the objects accessible to the user. Restricted contexts apply a second access check after users are granted access to the resource, to check the permissions of the running program as well. With extensions to existing authentication mechanisms, restricted contexts can also be applied across network connections to allow the use of network file systems. Finally, restricted contexts can be applied to uses other than untrusted code, such as for delegating authority between mutually trusting applications.

5.1 Restricted Contexts

Windows 2000 allows users to create a limited version of their access token, called a *restricted token*, which may access only a subset of the objects that the user may normally access. A process running with a restricted token is a *restricted context*, and its access rights are limited through three independent mechanisms. First, users may remove privileges so that the restricted context is limited to only access resources protected by access control lists. Second, users may disable groups, so that access granted to those groups does not apply to the restricted context. However, the groups must still be checked against ACEs denying access, so instead of removing the groups from the access token completely, they are instead marked USE_FOR_DENY_ONLY. Finally, and most powerfully, users may add a list of *restricted SIDs*, which represent the identity and access rights of the program being run and are used during access checks. Both the user's normal identities and the restricted SIDs must be granted access to an object. If either set of identities is denied access, then the access check fails. Restricted contexts can implement simple security policies, such as disabling administrative rights and privileges for most programs, as well as more restrictive policies such as limiting a program to accessing only a single file. This is accomplished by creating a restricted SID for the program and then setting an ACL that grants access to that SID on the desired file. When the program is run with the restricted SID in its restrictions, access checks on all objects except that file will fail, because no other ACL in the system grants access to the program. Figure 15 shows the algorithm for checking access with a restricted token.

```
BOOLEAN
RestrictedAccessCheck(Acl: ACL,
                      DesiredAccess : AccessMask,
                      RestrictedToken : AccessToken)

if (AccessCheck(Acl, DesiredAccess, RestrictedToken.PrincipalSids) &&
   (AccessCheck(Acl, DesiredAccess, RestrictedToken.RestrictedSids)
  return SUCCESS;
else
  return FAILURE;
```

Figure 15: The algorithm for checking access with a restricted token.

There are two broad approaches for choosing the restricted SIDs for a program. First, each program or class of similar programs may be assigned a different SID. The resources needed by those programs must grant that specific SID the required access. The second approach is to treat restricted SIDs as privileges protecting a class of resources, such as user interface objects or scratch file space, so that programs receive SIDs for each resource class they are allowed to access. The ACL on instances of a resource must grant access to the SID for that resource. The first approach provides tighter control, because restricted programs are only allowed to access specific objects. The second approach, though, simplifies administration by removing the need to identify every resource needed by a program. The two approaches may be combined, so that some resources are accessible through SIDs identifying the program while others are accessible through resources class SIDs. In addition to SIDs that are only used as restrictions, normal SIDs, such as the user's or a group's, may also be used in restrictions.

User SID	Jane User
Group SIDs	Administrators *(use for deny only)* Service Operators *(use for deny only)* Users
Restricted SIDs	StockTicker Restricted Windows
Privileges	(none)

Figure 16: A restricted token. In this token, the *Service Operators* group SID has been disabled so it can only be used to deny access, and all privileges have been removed. In addition, the *StockTicker* SID has been added to the Restricted SIDs field, so that SID must be granted access to any objects accessed by this token.

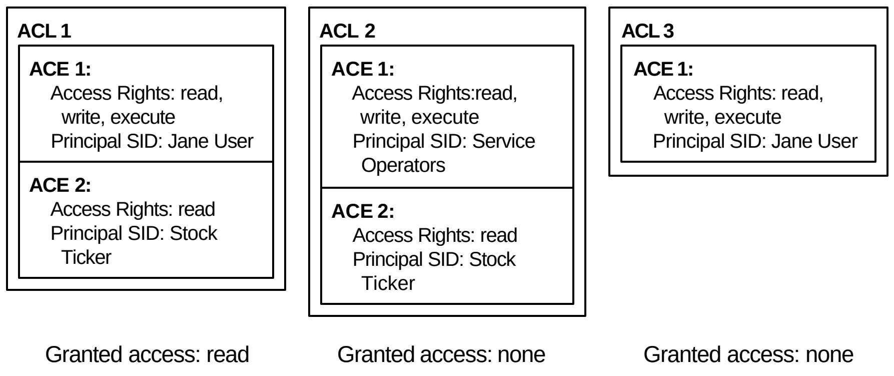


Figure 17: This example shows three ACLs accessed by the restricted context in Figure 16.

Figure 16 shows an example of a restricted context, and Figure 17 shows how it is used for an access check. The example access token has two restricted SIDs, '*StockTicker*', representing a downloadable stock-ticker application, and '*Restricted Windows*', granting access to the windowing system. In addition, all the privileges have been removed and one group, '*Service Operators*' has been disabled. The three ACLs shown in Figure 17 demonstrate the effect of restricted SIDs. In the first ACL, the restricted context is allowed read access, because both '*Jane User*' and '*StockTicker*' are granted access. With the second ACL, access is denied even though '*StockTicker*' is granted access because the '*Service Operators*' SID in the unrestricted portion of the token may only be used to deny but not to grant access. Similarly, the third ACL grants the restricted context no access, because there is no ACE granting '*StockTicker*' any access. As a result, the restricted context is granted access to only a subset of the resources available to the user. It is important to note that a restricted token cannot be used on a single thread to execute untrusted code, because that thread can access any process-wide resources, such as handles to open files, or stop impersonating and then run with an unrestricted token.

Restricted contexts can be used to implement the policy of *least privilege* [Saltzer and Schroeder 1975], which states that a program should have only the privileges necessary to perform its job and no more. Least privilege requires that the operating system know the set of resources a program requires, and then launch the program in a restricted context with access to just those resources. While Windows 2000 does not have a mechanism to describe the resources required by a program, it does provide the enforcement mechanism to limit resource access. With a proper policy in place, many common application exploits, such as macro viruses [CERT Coordination Center 1995], can be prevented because the application has no access to unrelated or unnecessary resources.

5.2 Applying Restrictions to Operating System Resources

Limiting access by placing restricted SIDs in access control lists works well for files. The file system stores ACLs persistently, so policy does not need to be specified each time the system restarts. Users also have tools to manipulate file system ACLs. Most importantly, the access rights on files (read, write, and execute) are the same rights that users want to limit for untrusted code. For these same reasons, though, many other system resources are difficult to protect with restricted SIDs:

1. Users do not control many objects internal to the operating system, so they may not have permission to modify the ACLs on those objects.
2. The operating system creates the ACL on many non-persistent objects, such as user interface objects, at boot time and users do not have an opportunity to store a new ACL on these objects.
3. The access rights for an operating system resource may be at the wrong granularity, such as in the case of network sockets, which do not distinguish between different endpoints.

Our solution to protecting operating system resources is twofold. First, we created several standard SIDs that may be used in restrictions to grant access to broad classes of system resources. The operating system uses these SIDs when protecting its own objects. For example, the “restricted-network” SID is used to grant access to network components, and the “restricted-windows” SID grants access to the user interface. Second, access to resources for which ACLs do not provide the correct granularity of protection must be denied by the operating system. In this case, the untrusted application contacts a trusted service in a separate process to perform the operation. For example, network client code contacts a trusted service that establishes network connections on its behalf. The service verifies the client’s identity and checks whether the untrusted code is allowed to contact the specific endpoint before creating the connection. Using a separate service violates our goal of enforcing access control with a single mechanism, but this problem can be reduced if the service itself uses ACLs to express the security policy. Unfortunately, the system-wide restricted SIDs for specific resources and the trusted service mechanism were dropped from Windows 2000. There is, though, a single system-wide restricted SID for identifying restricted contexts.

5.3 Remote Authentication with Restricted Contexts

Restricted contexts are most useful for local access control but unlike many sandboxing mechanisms, they may also extend across a network, such as to a network file server. Restricted contexts do not have access to a user's password, private keys for TLS [Dierks and Allen 1999], or Kerberos ticket cache [Neuman and T'so 1994], because the untrusted code could authenticate itself as the user and cause a remote server to build an access token without restrictions. Windows 2000, through the SSPI interface [Brown 2000] (similar to GSS-API [Linn 1993]), instead exposes only abstract authentication operations. This interface generates messages that the caller sends to a remote machine for authentication, but does not expose any secret data such as passwords. For example, the Kerberos protocol returns application request messages rather than tickets. The authentication protocol can include the restrictions in the authentication messages so that they are carried to remote servers. As long as the untrusted code is prevented from corrupting the restrictions, a restricted context may authenticate to any network service.

The Kerberos authentication protocol used in Windows 2000 [Neuman and T'so 1994] has a field, *authorization-data,* in its encrypted authentication messages with which the client can explicitly limit its authority on the server. When a restricted context attempts to authenticate, the Kerberos code captures the context's restrictions and stores them in the authorization-data field of a ticket. When a server receives the ticket, these restrictions are applied to the access token before the server application is allowed to impersonate the client. As a result, applications do not need to be aware that they are running with restrictions or that they are accessing a remote resource; instead the operating system manages transmitting the restrictions to the remote server. This feature was also implemented but not shipped with Windows 2000.

5.4 Limited Delegation with Restricted Contexts

Restricted contexts may also be used for application-level delegation of authority for applications that trust each other. It is common for Internet applications, such as web servers, to contact services running on other machines while processing a request. Windows 2000 normally requires a user's credentials, in the form of a password or Kerberos ticket to create an access token for the user. As a result, if the service wants to use the system access control routines, it must authenticate the client or be given access to an existing copy of the client's access token. Applications can access services on the same machine while impersonating the client, because the access token is copied through the kernel. However, applications cannot authenticate as the client to a service running on a separate machine unless the client's authentication protocol supports delegation of credentials and uses the same protocol as the application and server.

Restricted contexts provide an alternate mechanism for applications that trust each other to distribute the task of authorizing a client. Rather than requiring authentication protocols that support delegation, the application that authenticates a client instead captures the client's identity, group memberships, and privileges. When the application communicates with a remote service, it supplies those memberships and privileges as restrictions, either in its own protocol or using an authentication protocol as described in Section 5.3. This ability for an application to impersonate a client by restricting its own rights with the client's rights is called *limited delegation*. To implement this mechanism, the service must protect its resources with ACLs that, in addition to granting clients access, also grants the application full access. An access token for the application restricted by the client's identities then receives the client's correct access. Because the application receives full access to the server's resources, the service must trust the application both with all its resources and to correctly authenticate clients. However, limited delegation is applied only to the applications that trust each other, so the client does not need to trust the application with data outside the scope of the application, such as unrelated file servers.

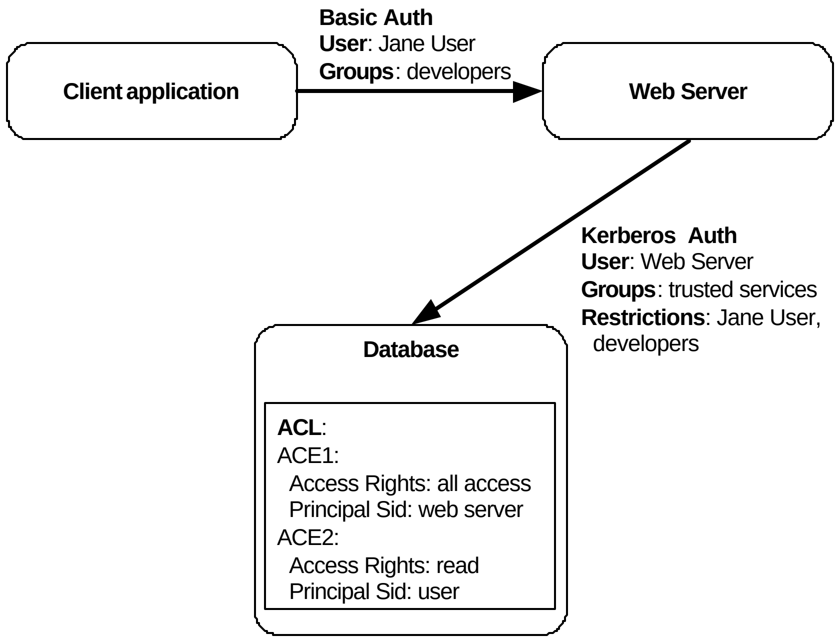


Figure 18: Limited delegation may be used by multi-tier applications to securely delegate the client's authority from the first tier server to later tiers by transmitting the client's identity as restrictions on the web server's identity.

Figure 18 demonstrates the use of limited delegation. The client authenticates with a web server using standard web authentication protocols, which causes the web server to build an access token. When the web server contacts a database using Kerberos authentication, it sends the security identifiers from the client's access token. The ACLs on the database grant the web server full access to all data and grant the client partial access. Access checks with the restricted token return the client's true access. This technique allows client identities to be forwarded between trusting servers without authentication protocol support for delegation, and without allowing the server to amplify their rights to those of the client and access resources unrelated to the application. Limited delegation also allows the use of a different authentication protocol between the client and the web server than between the web server and the database, because no client credentials are needed for the second hop.

5.5 Discussion

We do not have much experience with restricted contexts due to their limited implementation in the shipped version of Windows 2000. However, restricted contexts as they are implemented do allow users to limit their rights, so they need not run all programs with the same rights. Users may choose to run a web browser and mail program in a context without access to work documents. Instead of maintaining a separate account for administration, a simple program launch tool could let users disable their administrative access to the system when running normal programs and only enable it when running administrative tools. Allowing restricted contexts to be used for network authentication increases their utility because ordinary applications that need to access network services can be run in restricted contexts. As a result, restricted contexts are able to both provide safety from untrusted code and protect user data from attacks by subverted applications.

There is no policy component of restricted contexts in Windows 2000 that chooses the restrictions for a program. We implemented a policy based on Internet Explorer Zones [Microsoft Corp. 2001], which classify web sites into categories of trust and can also be used to assign policies to each zone. The policy classified executables by the DNS domain name of their source. This classification was used to select a restricted identity, which requires trusting DNS to translate names correctly names to addresses. The policies for code limited whether code is allowed network access, user interface access, and access to the user's data. This feature was dropped before Windows 2000 shipped. Another possible policy, used by WindowBox [Balfanz and Simon 2000], is to create several isolated environments with only limited sharing. Applications with similar security risks are run in the same environment and can share data freely. However, WindowBox limits sharing between environments to avoid the spread of viruses or Trojan horses. The advantage of this mechanism is that it presents a simple and understandable user interface. Restricted contexts could also be used in conjunction with the policy language used by MAPbox [Acharya and Raje 2000], which classifies applications according to their resource usage and provides parameterized categories of applications.

Beyond policy, there are several issues with restricted contexts that we have not resolved. We have not determined the correct context for a process executing code from multiple sources. Intersecting the two contexts may create too restrictive a context, and the union of the two contexts is not safe. The user interface for ACL editing also presents problems. Normally, all the SIDs that may be present on ACLs are stored on in the Active Directory. If users may fabricate SIDs for restrictions and place them on ACLs, then the ACL editor must have a mechanism for translating the SIDs into names, such as a separate database of locally defined SIDs or an external interface for translating SIDs. Despite these issues the restricted context mechanism remains a powerful tool for expressing many security policies.

6. RELATED WORK

The problems we faced for Windows 2000 are not unique and have been addressed by many earlier systems, although not in the same combination. Other directory services support fine-grained access control, and the inheritance of ACLs has been addressed in many settings ranging from object-oriented databases to distributed systems. Restricting the access rights of programs has also been addressed by many operating systems. In this section we discuss relevant systems and their relationship to our design.

6.1 Fine Grained Access Control

While there have been many access control list implementations in operating systems, they typically cannot support directory services. Instead, directory services implement their own mechanisms in order to support complex objects. The access control model in Novell’s directory service, NDS 8 [Cadjan and Harris 1999], resembles the model in Windows 2000 due to its similar application domain. NDS supports both inheritance of access rights as well as protection of individual properties. NDS differs from the Active Directory by implementing dynamic inheritance, in which access rights on containers are evaluated during access rather than propagating the rights to individual objects. Furthermore, NDS does not support grouping properties into property sets, and does not support the inheritance of rights for specific properties. Instead, only rights for all properties at once may be inherited. NDS’s *inherited rights filters*, which block the inheritance of specific rights (such as read all properties) instead of all rights, are more flexible than Windows 2000's mechanism that blocks all inheritance.

The access control mechanism in DCE [Mackey and Salz 1993] also supports directory services. DCE stores many separate ACLs to expand the rights available for an object, while Windows 2000 incorporates the additional rights into a single list. Also, DCE stores two separate ACLs for inheritance, one for newly created objects and one for containers, rather than distinguishing between many different types of objects within a single ACL.

The Netscape Directory Server [Sun Microsystems 2001] uses ACL rules rather than explicit ACLs on objects. The rules contain a *target*, which is an LDAP search rule [Yeong et al. 1995], *permissions*, such as read and write, and a *bind rule*, which indicates the clients to which the ACL applies. This format is more expressive than the ACLs in Windows 2000, because the target field may specify not only a single object or tree of objects but also arbitrary sets of objects based on their properties. However, these rules are specific to LDAP and cannot be used for other applications, and are therefore not as general as ACLs in Windows 2000.

The access control list support from other operating systems, such as Sun's Solaris [Winsor 2001] and Linux [Grunbacher 2001] is not as flexible as that in Windows 2000 in that they are designed exclusively for file system use. Both the Solaris and Linux ACL mechanisms store a single default ACL on directories for all files created within the directory, so they are unable to distinguish between multiple types of objects. In addition, the ACLs do not store inheritance information, so changes cannot be propagated through a tree without losing existing ACLs. Security-Enhanced Linux [Loscocco and Smalley 2001b] enhances ACLs with a class identifier, to allow different rights for different types of objects, such as TCP sockets and raw sockets. However, this ACL support neither expands the number of rights for a single object nor specifies how ACLs are inherited hierarchically.

6.2 Access Control Inheritance

The issue of inheriting ACLs hierarchically has been addressed in many settings. Twidle and Sloman [1988] discuss the inheritance of rights between domains of objects in a distributed system, and specify, similar to Windows 2000, that both positive and negative rights in a subdomain should override the inherited rights from a parent domain. Moffett et al. [1990] discuss mechanisms for implementing inheritance statically by combining the inherited access control entries from parent containers into the ACL for a single object, which is our choice for Windows 2000. [Fernandez et al. 1989] addresses the issue of inheritance in an object oriented database, and outlines an inheritance policy similar to Windows 2000, but implemented dynamically by inspecting the ACLs of parent objects rather than pre-computing the ACL on an object.

WebDAV [Clemm et al. 2001] uses a similar inheritance model to Windows 2000, reflecting Microsoft's input into its design. In addition to supporting static inheritance and protecting objects from inheritance, WebDAV does not restrict objects to a tree structure. Instead, ACLs explicitly reference the source of their inherited entries.

6.3 Restricting Executables

The problem of restricting executing code has also been addressed by many other systems, although in a different fashion. Most similar to restricted tokens are the *process access groups* of the Andrew distributed system [Satyanarayanan 1989], in which a child process can be launched with some of its groups disabled. While the groups were also disabled when accessing network services, the system does not address negative authorizations, in which users could be denied access based on group membership. Process access groups also do not allow for finer grained control than existing security groups.

Similar to restricted contexts, Janus [Goldberg et al. 1996], Tron [Berman et al.], and MAPbox [Acharya and Raje 2000] provide isolation for untrusted code by protecting objects in the operating system. However, rather than protecting the objects directly, they instead trap system calls and inspect the parameters for access. In addition, Janus provides a language, also used by MAPbox, to specify security policies. MAPbox enhances Janus by providing parameterized behavior classes, so that applications with similar needs may share policies. These approaches use a separate set of security mechanisms and configuration tools for protecting users from untrusted code than are used for protecting users from each other, and as a result are not integrated with the existing operating system security mechanisms. The benefit of these systems is that they provide a policy for restricting code and are more flexible than restricted contexts because they see all the parameters to system calls, rather than just the desired access to an object. The Linux Intrusion Detection System (LIDS) [Hatch 2001] provides enhanced isolation functionality to Linux, and allows rights to be granted to programs rather than just users, but again is not integrated with other operating system protection mechanisms. The WindowBox project [Balfanz and Simon 2000] provides a policy for isolation by separating applications onto distinct user-visible desktops rather than just running them in different contexts. However, rather than having a general mechanism for limiting executable code, WindowBox limits access by tagging objects with a single SID, and then checking for that SID in an access token.

Several operating systems have been constructed to limit the damage from exploited programs by providing additional isolation between processes. Hewlett Packard's Compartmented Mode Workstation (CMW) [Zhong 1997] and Domain and Type Enforcement (DTE) in Unix [Walker et al. 1996] provide isolation between processes and restrict the objects accessible to a process based on their type. While the security models of these operating systems are more powerful than Windows 2000's, these operating systems also required greater development effort to achieve that power.

Security-Enhanced Linux [Loscocco and Smalley 2001a] also restricts programs by labeling objects and provides automatic protection domain changes when invoking a new program. This operating system, similar to restricted contexts, can also restrict program to accessing only a small set of objects, although it augments the operating system protection with rules in a configuration file rules instead of storing access control lists on objects. Again, the guarantees of SELinux are stronger, but the changes to the operating system are greater, and the operating system is left with multiple mechanisms for expressing access control. However, unlike Windows 2000, SELinux, CMW and DTE in Unix provide better isolation of programs because they have a notion of information flow [Bell and LaPadula 1976, Denning 1976]. In Windows 2000, a rogue mail program in a restricted context may save a file that is later accessed from an unrestricted context where it can cause damage, whereas these operating systems label objects with their source to fully is olate programs and their outputs.

Restricted contexts are a mechanism that may by used to implement many security policies, such as role-based access control (RBAC) [Sandhu et al. 1996] in which users select specific roles when performing job tasks. A user may have different access rights depending on their role when running a program. Similar to RBAC, restricted contexts allow programs to be run with different rights according to their task. However, restricted contexts depend on the existence of an unrestricted context that has complete access to the user's resources, while role-based access control does not.

Restricted contexts are similar to the compound principals from [Abadi et al. 1993], where two principals can be required for access. Compound principals, though, are used in ACLs to grant access to the combination of two subjects, such as "Jane User and StockTicker". Restricted contexts instead subdivide an existing subject of access control into two, and require that both parts be granted access separately.

Finally, Mazières and Kaashoek [1997] suggest that operating systems should support *hierarchically named capabilities*, in which a user may append identifiers to her user identifier to create many levels of sub-identities. These capabilities are similar to restricted contexts in that users can create limited versions of their identity, but programs must specify which single capability is to be used for each access.

7. CONCLUSION

In this paper we presented the extensions made to the Windows NT 4.0 access control mechanisms for Windows 2000. These extensions enable the access control mechanisms of Windows NT, designed primarily for file systems, to apply to applications with more complex needs, such as a directory service. While many of the ideas have been seen before in other applications or systems or in slightly different forms, in Windows 2000 the same implementation of ACLs is used by all system services and many applications rather than creating a separate mechanism for each use. The combination of features in Windows 2000's ACLs provides a balance of feasibility, performance, and manageability. In particular, extending access control entries to specify both a portion of an object for access checks and a type of object for inheritance allows the existing model, designed for file systems, to be applied to many other applications. The extended inheritance controls enable centralized management of large hierarchies of objects by allowing inheritance to be reapplied without disrupting previously modified ACLs. The addition of restricted contexts makes it possible to apply operating system security mechanisms to isolate misbehaving code by allowing users to restrict the set of objects accessible to a program. Unfortunately, the improvements described in Sections 5.2 and 5.3 (protecting system objects and remote authentication, respectively) did not make it into the shipped version of Windows 2000. Overall, these changes greatly improve the scalability and security of the Windows 2000, while retaining the simplicity of a single access control mechanism throughout the operating system.

ACKNOWLEDGEMENTS

We would like to thank the other members of the security, directory service, and DCOM teams at Microsoft, in particular Robert Reichel and Murli Satagopan, for contributing to the ideas presented here. Robert Grimm was our fabulous shepherd on an earlier version of this paper and provided comments on several versions. Steve Gribble and Andrew Whitaker provided valuable feedback on an early draft of this paper. Kedar Dubhashi provided valuable updates on the changes made for Windows XP.

REFERENCES

ABADI, M., BURROWS, M., LAMPSON, B., AND PLOTKIN, G. 1993. A Calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems 15*, 4(Oct), 706-734.

ACHARYA, A. AND RAJE M. 2000. MAPbox: Using parameterized behavior classes to confine untrusted applications. In *Proceedings of the 9th USENIX Security Symposium*, Denver, Colorado, Aug..

ANDERSON, J. 1972, Computer security technology planning study. Technical Report ESD-TR-73-51, Vols. I and II, Air Force Electronic Systems Division. NTIS document number AD758206.

BALFANZ, D. AND SIMON, D. 2000. WindowBox: A simple security model for the connected desktop. In *Proceedings of the 4th USENIX Windows Systems Symposium*, Seattle Washington, Aug.

BELL, D. AND LAPADULA, D. 1976. Secure computer systems: Unified exposition and Multics interpretation. Technical Report MTR-2997 Rev. 1 (Mar.). MITRE Corp., Bedford, MA. Also ADA023588, National Technical Information Service.

BERMAN, A., BOURASSA, V., AND SELBERG, E. 1995. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the 1995 USENIX Winter Technical Conference*, New Orleans, Louisana, Jan., 165—175.

BROWN, K. 2000. Explore the security support provider interface using the SSPI workbench Utility. *MSDN Magazine*, Aug. Available at http://msdn.microsoft.com/msdnmag/issues/0800/ Security/Security0800.asp.

CADJAN, N AND HARRIS, J. 1999. Administering NDS, corporate edition. McGraw-Hill Professional Publishing.

CALLAGHAN, B., PAWLOSKI, B. AND STAUBACH, P. 1995. NFS version 3 protocol specification. Request for Comments RFC 1813, Internet Engineering Task Force.

CERT COORDINATION CENTER, 1995. CERT Advisory CA-2000-16 *Microsoft 'IE Script'/Access/OBJECT Tag Vulnerability*. Aug. Available at http://www.cert.org/advisories/CA-2000-16.html.

CLEMM, G., HOPKINS, A., SEDLAR, E., AND WHITEHEAD, J. 2001. WebDAV access control protocol. Internet draft draft-ietf-webdav-acl-07, Internet Engineering Task Force, Nov.

DENNING, A. 1997. ActiveX controls inside out, second edition, Microsoft Press.

DENNING, D. 1976. A Lattice Model of Secure Information Flow. *Communications of the ACM 19*, 5(Aug.), 236-243.

DIERKS, T. AND ALLEN, C. 1999. The TLS protocol. Request for Comments RFC 2246, Internet Engineering Task Force, Jan.

EDDON, G. AND EDDON, H. 1998. Inside distributed COM. Microsoft Press.

FERNANDEZ, E., GUDES, E., AND SONG, H. 1989. A security model for object-oriented databases. In *Proceedings of the IEEE Symposium on Security and Privacy.* Oakland, California, May, 110-115.

GOLDBERG, I., WAGNER D., THOMAS, R. AND BREWER, E. A. 1996. A secure environment for untrusted helper applications - Confining the Wily Hacker. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, California, Jul.

GRUNBACHER, A. 2001. Extended attributes and ACLs for Linux. Available at http://acl.bestbits.at.

HATCH, B. 2001. An overview of LIDS, part one. Oct. Available at http://www.securityfocus.com/ infocus/1496.

ISEMINGER, D. 2000. *Active Directory Services for Microsoft Windows 2000 Technical Reference*. Microsoft Press

LEACH, P. AND SALZ, R. 1998. UUIDs and GUIDs. Internet Draft draft-leach-uuids-guids-01.txt. Internet Engineering Task Force, Feb.

LINN, J. 1993. Generic Security Service API. Request For Comments RFC 1508, Internet Engineering Task Force, Sep.

LOSCOCCO, P., AND SMALLEY, S. 2001a. Integrating flexible support for security policies into the Linux Operating System. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, Boston Massachusetts, June.

LOSCOCCO, P., AND SMALLEY, S. 2001b. Meeting critical security objectives with security-enhanced Linux. In *Proceedings of the 2001 Ottawa Linux Symposium*. Ottawa, Ontario, Jul.

MACKEY, D. AND SALZ, R. 1993. DCE ACL Library – Functional Specification. OSF DCE SIG Request For Comments 46.0, Oct.

MAZIÉRES, D. AND KAASHOEK, M. F. 1997. Secure applications need flexible operating systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems*, Cape Cod, Massachusetts, May.

MICROSOFT CORPORATION. 2000. Novell wrong about windows 2000 security hole. Feb. Available at http://www.microsoft.com/WINDOWS2000/server/evaluation/news/bulletins/ novellresponse3.asp.

MICROSOFT CORPORATION. 2001, Internet security part 1: the basics. *Microsoft Insider*. Available at http://www.microsoft.com/insider/internet/articles/security.htm.

MICROSOFT KNOWLEDGE BASE. 2000. Large numbers of ACEs in ACLs impair directory service performance. Available at http://support.microsoft.com/support/kb/articles/q271/8/76.asp, Sep.

MOFFETT J., SLOMAN, M., AND TWIDLE, K. 1990. Specifying discretionary access control policy for distributed systems. *Computer Communications 13*, 9(Nov.), 571-580.

NEUMAN, B. C. AND T'SO, T. 1994. Kerberos: An authentication service for computer networks. *IEEE Communications 32*. 9(Sept.), 33-38.

NOVELL INC. 2000, The NDS advantage: AD security. Feb. Available at http://www.novell.com/competitive/nds/security.html.

RITCHIE, D. AND THOMPSON, K.1974. The UNIX timesharing system. *Communications of the ACM 17,* 7(Jul.) 365-375.

SALTZER. J. AND SCHROEDER, M. 1975. The Protection of Information in Computer Systems. *Proceedings of the IEEE 63*, 9(Sep.), 1278-1308.

SANDHU, R., COYNE, E., FEINSTEIN, H., AND YOUMAN, C. 1996. Role-based access control models. *IEEE Computer 29*, 2(Feb.) 38- 47.

SATYANARAYANAN, M. 1989. Integrating security in a large distributed system. *ACM Transactions on Computer Systems 7*, 3(Aug), 247-280.

SUN MICROSYSTEMS INC. 2001. Deployment Guide, Netscape Directory Server version 6.0. Available at http://enterprise.netscape.com/docs/directory/index.html

TWIDLE, K., AND SLOMAN, M. 1988. Domain based configuration and name management for distributed systems. In *Proceedings of the IEEE Workshop on the Future Trends of Distributed Computer Systems in the 1990s*, Hong Kong, Sep., 7-153.

WALKER, K., STERNE, D., BADGER, M., PETKAC, M., SHERMANN, D. AND OOSTENDORP K. 1996. Confining Root Programs with Domain and Type Enforcement (DTE). In *Proceedings of the 6th USENIX Security Symposium*, San Jose, California, Jul.

WINSOR, J. 2001. Solaris 8 System Administrator's Guide, Prentice Hall.

YEONG, W., HOWES, T., AND KILLE, S. 1995. Lightweight Directory Access Protocol. Request for Comments RFC 1777, Internet Engineering Task Force, Mar.

ZHONG, Q. 1997. Providing Secure Environments for Untrusted Network Applications. In *Proceedings of the 2nd IEEE International Workshop on Enterprise Security,* Cambridge, Massachusetts, June.