Improving the Granularity of Access Control in Windows NT

Michael M. Swift[†], Peter Brundrett, Cliff Van Dyke, Praerit Garg, Anne Hopkins, Shannon Chan, Mario Goertzel, Gregory Jensenworth

Microsoft Corporation
1 Microsoft Way
Redmond, WA 98052
mikesw@cs.washington.edu,
{petebr, cliffv, praeritg, annehop, shannonc, mariogo, gregjen} @microsoft.com

ABSTRACT

This paper presents the access control mechanisms in Windows 2000 that enable fine-grained protection and centralized management. These mechanisms were added during the transition from Windows NT 4.0 to support the Active Directory, a new feature in Windows 2000. We first extended entries in access control lists to allow rights to apply to just a portion of an object. The second extension allows centralized management of object hierarchies by specifying more precisely how access control lists are inherited. The final extension allows users to limit the rights of executing programs by restricting the set of objects they may access. These changes have the combined effect of allowing centralized management of access control while precisely specifying which accesses are granted to which programs.

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

Keywords

Access Control Lists, Windows 2000

1 INTRODUCTION

The Windows NT 4.0 operating system provides a secure platform for applications by supporting authentication, authorization, and auditing. However, the addition of the Active Directory in Windows 2000 and the increasing frequency of security exploits from buggy applications demonstrated several limitations of Windows NT security. The Active Directory, a hierarchical directory service [17], requires access control at a finer granularity than can be supported by the mechanisms in Window NT 4.0. Another weakness is that users cannot 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 Active Directory in Windows 2000 is a directory service, which "provides a place to store information about network-based entities, such as applications, files, printers, and people" [17]. It stores a database of many types of objects, including printers, services, and users. The objects in the directory each have many data properties, such as name, location, and owner. Every object has a unique name, and the objects are organized as a tree-shaped hierarchy. Containers are used to group objects, and there may be objects of many different types within a single container. While the directory service is installed with many predefined object types, it also allows new object types to be defined and existing objects types to be extended with new properties. The primary task of the Active Directory is to answer queries, which range from retrieving all the properties of a single object to returning a subset of the properties of all the objects matching a query expression.

The major access control mechanism in Windows NT 4.0 is the access control list (ACL). These lists are used throughout the operating system and by many applications. The access control list data structures and algorithms were designed for services, such as the file system, with only a few types of objects (files and directories), and with only a small number of operations (such as read, write and execute) and data properties (such as owner and creation time). However, the Active Directory contains hundreds of types of objects and every object has many properties that must be protected separately. As a result, we discovered three limitations of the access control mechanisms in Windows NT 4.0: the mechanisms do not support large numbers

[†] This work was performed while this author was employed at Microsoft Corporation. The author's new address is: University of Washington, Seattle WA 98195.

of properties or operations on an object, having many different object types within a container, or propagating changes to ACLs through a tree of objects.

The solution for Windows 2000 to all three problems is to extend the entries in access control data structures with the type of object and the scope to which they apply. For example, access control list entries in Windows 2000 specify what type of object is being controlled, what types of objects may be created, and what types of objects will copy the entry into their access control list upon creation. As a result, access control lists can precisely specify what objects may be accessed as well as how to propagate that information through a large hierarchy of objects.

In addition to not supporting fine-grained access control lists, Windows NT 4.0 also does not support fine-grained subjects, with which a user can specify the rights of a particular program. For example, downloaded ActiveX controls [20] have to be trusted with the full rights of the user and cannot be restricted to accessing only specific objects. In addition, bugs in applications, such as web browsers and email clients, can inadvertently expose all the files on a computer to an attacker [6]. Using *restricted contexts*, Windows 2000 allows programs to run with limited authority, so they may only access certain objects. As a result, users can implement the principle of *least privilege*, in which programs are only granted access to the objects necessary for their execution.

This paper presents the access control mechanisms in Windows 2000, with an explanation of what tradeoffs were made in the design. As background, in Section 2 we present a description of access control in Windows NT 4.0. This is followed by a description of object-specific access control lists in Section 3 and inheritance improvements in Section 4. In Section 5 we describe changes to limit the rights of untrusted code. In Section 6, we discuss related work and then conclude in Section 7.

2 ACCESS CONTROL MECHANISMS IN WINDOWS NT

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. The security models of both Windows NT 4.0 and Windows 2000 separate *objects*, entities being accessed, from *subjects*, the active entities performing accesses. Access control in both systems is based on *access control lists*, which, for each object, specify the access granted to different subjects.

2.1 Subjects

A subject in Windows NT is represented by an *access token*, which is a kernel object encapsulating a user's identities and privileges. Users may be organized into groups, such as all users that work on a project together. *Security identifiers*, or SIDs, are variable-length byte strings that represent security principals, such as users or groups of users. 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 objects for backup/restore 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. Access tokens, shown in Figure 1, are constructed during logon by a trusted *authentication package*, which is responsible for authenticating the user and determining which user identifier, group identifiers, and privileges should be in the access token. Access tokens are associated with both processes and threads. If a thread has no access token, then the access token from the containing process is used for access control. Each process is assigned an access token when it is created, and, after creation, the access token may not be replaced or changed except for enabling or disabling privileges. Threads, though, may have their access tokens replaced at any time, allowing programs to perform specific actions with different identities.

```
User SID:       Jane User
Group SIDs:     Administrators
                Service Operators
                Users
Privileges:     Start Service
                Load Device Driver
                Shut Down
```

Figure 1: Example of a simplified access token in Windows NT 4.0, containing user and group identities and privileges.

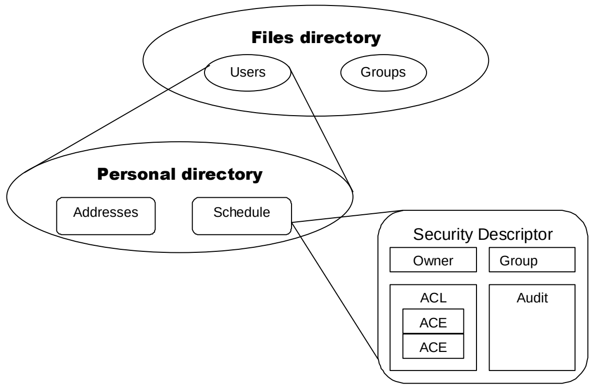

Figure 2: Every object, such as directories and files has a security descriptor. A security descriptor has an owner, group, ACL, and auditing information.

2.2 Access Control

Every object in Windows NT that needs access control, such as files, processes, and devices, is assigned a *security descriptor*. These descriptors store all the security state for an object, consisting of the owner, group (similar to the Unix owning group field), access control list, and auditing information. The operating system provides a standard representation of access control lists (ACLs) for use by both system services and applications. If a security descriptor has no ACL, then it is assumed that all users are granted full access to the object. Figure 2 shows a simple file system and the security structures associated with files and directories.

An access control list is a container for *access control entries* (ACEs), which determine which access rights should be granted or denied to principal. An ACL may contain an arbitrary number of ACEs for different users or groups of users. The two types of ACEs in Windows NT 4.0 are ACCESS_ALLOWED_ACEs,

```
ACL Header:
  Revision: version 1
  ACL Size: 100 bytes
  ACE Count: 2
ACE 1
  Header:
    Type: ACCESS_ALLOWED_ACE
    Flags: OBJECT_INHERIT
  Access Rights: read, write, execute
  Principal SID: administrators
ACE 2
  Header:
    Type: ACCESS_ALLOWED_ACE
    Flags: CONTAINER_INHERIT
  Access Rights: read
  Principal SID: everyone
```

Figure 3: Example access control list (ACL) with two entries (ACEs).

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

Table 1: Flags in the ACE_HEADER that control inheritance.

which grant a principal access, and ACCESS_DENIED_ACEs, which deny access. A sample ACL is shown in Figure 3. The header field identifies the type of ACE and contains flags controlling how it is inherited. 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 4.0 allows sixteen bits to be defined by the implementer of an object for specific access rights, such as reading or writing data for a file, or listening on a socket and reserves sixteen bits for operating system use.

The *security reference monitor* in the Windows NT kernel is responsible for correctly evaluating ACLs for all applications and system components. Applications call the *AccessCheck* routine to make an access control decision. This 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. 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. Interleaving allow and deny ACEs allows Windows NT to emulate Unix file system ACLs [22] by ensuring that only one ACE grants access to a subject. For example, by containing an ACE granting a group access to a file followed by an ace denying the group all other accesses, the ACL ensures that a member of that group receives just the granted accesses and no more. However, the ordering convention in Windows NT is to place ACEs denying access before ACEs granting access, so that deny entries take precedence. Only if all the requested access rights are granted, or if the application requested the maximum allowed access and some right was granted, does the access check succeed.

While there are routines to directly modify existing access control lists, ACLs in Windows NT 4.0 are usually created by copying entries from the container of an object when it is created. When an object is created in a container, such as a file in a directory, some of the entries from the container's ACL are *inherited* onto the new object, and the corresponding ACEs are copied into the new object's ACL. Inheritance of ACEs is controlled through the flags in the ACE_HEADER structure, shown in Table 1. ACEs are *effective* if they apply to the object being protected; otherwise they are marked INHERIT_ONLY and are used for inheritance. ACEs may be marked as inherited onto *containers* (objects that can contain other objects), inherited onto *objects* (objects that cannot contain other objects), both, or neither. All inheritable access control entries are inherited onto new containers, so the ACEs can be propagated to objects created within that container. However, if the ACE was not marked CONTAINER_INHERIT, then the INHERIT_ONLY_ACE flag is set. As a result, the ACE is not effective and has no impact on access to the container. ACL inheritance allows permissions set on a directory to be propagated to every new file and directory created underneath it, but has no impact on already existing objects.

2.3 Limitations of Windows NT 4.0 Access Control

The Windows NT access control structures and mechanisms are powerful and flexible, and can be used to emulate other forms of access control lists, such as Unix [22] and DCE [15]. However, the mechanisms have several limitations:

- Access masks are only sixteen bits, so a single ACL can only control sixteen different access types.
- Inheritance does not distinguish between types of objects with different access rights.
- Propagating access control changes to a tree of objects may be ambiguous if some of the objects have ACLs other than the one inherited, because the inherited ACL and explicitly specified one must be merged.
- There is no mechanism for restricting the rights of a process other than disabling privileges.

The first three flaws are most apparent in the context of the Active Directory. The Active Directory supports many types of objects, such as users, groups, services, and machines, as well as many types of containers. Every object has of a set of properties, each of which may need to be protected differently. The Active Directory also requires that permissions be administrable from the top of the directory hierarchy so that an administrator may grant control of certain types of objects or groups of objects to another administrator. The fourth 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 result, programs have to be prevented from unnecessary access to user and system resources. These limitations forced us to update the access control mechanisms in Windows NT 4.0.

3 OBJECT-SPECIFIC ACCESS CONTROL

The Active Directory could not be supported by the Windows NT 4.0 access control infrastructure for three reasons: (1) the number of object types, (2) the number of properties on each object, and (3) the ability to add new object types and new properties to existing objects. The large number of object types requires a different access right to control creation of each type. The large number of properties on each object requires more access rights than the sixteen provided. The ability to create new objects requires dynamically extending the number of access rights.

We considered storing a separate ACL for each property on an object. However, accessing multiple properties simultaneously would then require multiple calls to *AccessCheck* and would create unnecessary overhead when many properties have the same protection. Another possibility we considered was to extend the access mask format so that more than sixteen bits are used to represent rights. However, a bit field would have been difficult to manage when properties were 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 portion of the object or type of object to which the ACE applies.

3.1 Object Types in ACEs

The new ACE format introduced in Windows 2000 adds two fields to each entry. An example of the new ACCESS_ALLOWED_OBJECT_ACE structure is shown in Figure 4. The first new field, *ObjectType*, identifies the type of object or property to which the ACE applies. The second new field, *InheritedObjectType*, controls which types of objects inherit the ACE and will be discussed in Section 4. Both types are represented as GUIDs [13], which are sixteen-byte values used by DCOM [19] to identify object classes.

The *ObjectType* field extends the set of rights available for an object. Applications can supply an object-type GUID to the new *AccessCheckByType* routine and only ACEs that either have no object type or the same object type are evaluated. As a result, the set of available access rights is expanded because the object-type GUID distinguishes between different sets of rights. In addition, applications may add new object-type GUIDs for an object dynamically, so the set of access rights can be expanded. The Active Directory associates a GUID with each object property on each type of object, and the same set of access rights, such as read and write, are used for each property. The directory service can then check for access to a property by passing the requested access and the GUID of the property to *AccessCheckByType*. The Active Directory also uses object-specific access control to extend container rights, such as the right to create a child object. The object type for such a right identifies the type of object that may be created. As a result, an administrator can explicitly allow a user to create certain types of objects but not others.

```
ACE:
  Header:
    Type: ACCESS_ALLOWED_OBJECT_ACE
    Flags: OBJECT_INHERIT
  ObjectType: LogonScriptPath
  InheritedObjectType: users
  Access Rights: read, write
  Principal SID: administrators
```

Figure 4: Example object type specific ACE granting administrators access to the logon script path. This ACE is inherited onto user objects. The new fields are shown in boldface.

```
Level 0: {none}
  Level 1: {GUID for profile property-
            set}
    Level 2: {GUID for home directory
              property}
    Level 2: {GUID for logon script
              property}
  Level 1: {GUID for public properties}
    Level 2: {GUID for user name}
    Level 2: {GUID for user description}
```

Figure 5: A multi-level list of object-type GUIDs for an access check.

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. The number of calls to check access also increases when multiple properties are accessed. The cost of access control in Windows 2000 is reduced by, first, allowing multiple object-type GUIDs to be passed to access check routines and, second, by allowing properties to be grouped into property sets. Property sets are identified by a GUID, and access to a property is granted if the access is granted by the GUID of its property set. As a result, the set of rights for each element of the property set is the same. Applications must still request access to each property individually rather than requesting access to the property set as a whole, because a subject may be granted access to each property separately by ACEs specifying only a property.

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 the new access control routines in Windows 2000, such as *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 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. An example of such a list is shown in Figure 5. Each GUID in the list has a hierarchy level that indicates whether the GUID corresponds to an object type, a property set, or a property. The reference monitor performs a separate access check for each property requested. Only access control entries with no object type GUID or an object type matching the property or its property set are evaluated.

3.3 Example

An example of how object types in ACEs are used is shown in Figure 6. The example shows an ACL on a user object in the Active Directory service. The first ACE grants administrators full control over all the properties of a 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. Changing a password does not correspond to reading or writing a property. Instead, an object type is being used as an *extended right*, because it extends the number of access rights for a single object to cover an additional operation rather than protecting a specific property. As a result, the required access is *control* rather than read or write. In this case, the Active Directory understands that the password change protocol, such as kpasswd [25], is allowed to change a user's password if it can prove knowledge of the user's previous password.

3.4 Discussion

The primary purpose of object-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 lessens the memory and performance impact of a large set of rights by allowing many rights to be coalesced into a single access control entry. The Active Directory uses these features to organize properties with similar access control requirements, such as all the contact information properties, into property sets.

There has been some customer resistance to specifying access control on individual properties: in particular, the large number of properties can be difficult for administrators to manage, so access control changes are usually applied at the property set level. One possible solution would be higher-level management tools that provide better abstractions while manipulating per-property access control entries. There has also been resistance by administrators to dividing the administration of an object between multiple individuals. However, this may be an artifact from Windows NT 4.0, in which each application stored its information separately, removing the need to share access to objects. As more applications use the Active Directory, shared access to objects and split administration should become more common.

One downside of using property sets is the inexact match between membership in property sets and administrative control. For example, when upgrading Windows NT 4.0 servers to Windows 2000 the access control lists protecting user objects in the Windows NT 4.0 directory service are converted to object-specific ACEs. However, each access right for a user object in Windows NT 4.0 grants access to many properties. These groups of properties do not map perfectly onto property sets in Windows 2000, so rather than converting an access right in Windows NT 4.0 into access to a Windows 2000 property set, it is instead converted into a sequence of ACEs granting access to each property. The resulting access control lists can be many times longer than those on user objects created natively in Windows 2000. Similarly if an administrator wants to grant access to most of the properties in a property set but not all, she must explicitly name all the 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.

```
ACL Header:
  Revision: version 2
  ACL Size: 100 bytes
  ACE Count: 3
ACCESS_ALLOWED_ACE
        Principal: Administrators
        Access: read, write, delete, control
ACCESS_ALLOWED_OBJECT_ACE
        Principal: Group Admins
        Access: read,write
        ObjectType: {GUID for public
                     property-set}
ACCESS_ALLOWED_OBJECT_ACE
        Principal: Jane User
        Access: control
        ObjectType: {GUID for change-
                     password}
```

Figure 6: A sample ACL using object-type specific access control entries.

Despite these drawbacks, object-specific access control is crucial to the Active Directory. Identifying properties by GUIDs simplifies adding and removing properties from an object by leaving ACEs for other properties unchanged. In addition, administrators can grant access to just the properties needed for a job function or application rather than all those controlled by the same access right.

4 INHERITANCE CONTROL

Windows NT 4.0 assigns access control to new objects primarily through inheritance of access control entries from the access control lists on containers. There are two primary limitations to the inheritance mechanisms in Windows NT 4.0:

- If containers can hold multiple types of objects, it is impossible to specify that different access control lists be inherited onto each object type.
- It is difficult to propagate changes to ACLs through a tree of objects, because inheritance rules cannot be re-applied without erasing any modified ACLs lower in the tree.

Both problems arise in the Active Directory, which has a large hierarchy of containers and where there are many object types. One of the goals for the Active Directory is to allow *delegation*, so that one administrator can grant another administrator control over a subset of the objects in the directory service, such as all those of a particular type or in a particular sub-tree. As a result, administrators must be able to change permissions at the top of the directory and let the effects propagate down.

We considered several solutions for each problem. One solution for supporting multiple object types, similar to the design of ACLs in DCE [15], is to store multiple ACLs on a container, one for each object type. However, that approach may be inefficient if many entries are common to all child objects. In addition, the routines for manipulating access control data in Windows NT 4.0 do not support multiple ACLs on a single object. Another option is to store default ACLs for the various object types in a separate database. However, this solution 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.

To allow granting rights to a tree of objects, we considered the *dynamic inheritance* approach, as used in the Novell Directory

Service [21]: 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. 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 [5]. Similarly, the Active Directory stores data in a flat database, so there is no convenient opportunity to access the ancestral containers while performing a query. Dynamic inheritance would require first locating the object and then locating all of its ancestors. Furthermore, object accesses and queries are relatively frequent, while changes to ACLs are infrequent. It is therefore more efficient to propagate ACLs when they are changed than to perform additional checks for each access. In addition, most objects inherit their access control lists, so, in the common case, the ACL determining access to an object will be stored on one of its ancestors. As a result, our design uses *static inheritance*, in which inheritance is reapplied only when ACLs change and only a single ACL is evaluated during access checks.

4.1 Type-Specific Inheritance

Similar to object-specific access control, Windows 2000 allows type-specific inheritance. The CONTAINER_INHERIT and OBJECT_INHERIT flags from Windows NT 4.0 are restricted to specific object types by adding a field, *InheritedObjectType*, that identifies the type of object that may inherit the ACE. The rules for determining which ACEs are inherited now require two steps: first, find ACEs in the container's ACL with OBJECT_INHERIT set if an object is being created and CONTAINER_INHERIT set if a container is being created. Second, discard ACEs with an *InheritedObjectType* field that does not match the type of the new object or container. As in Windows NT 4.0, all inheritable ACEs are propagated to containers for future inheritance. The result of this change is that different types of objects can inherit different sets of ACEs from a container, and all objects inherit ACEs that are not restricted by inherited object types.

An example of type-specific inheritance is shown in Figure 7. In this example, the *services* container has ACEs that are to be inherited onto all objects, *Web Service* objects, and *RPC Service* objects. The *WWW* service, a Web Service, inherits the ACEs with no inherited object type and with a Web Service object type. Similarly, the *Names* service inherits the first ACE and the RPC Service ACE. This example demonstrates the power of type specific inheritance: a single ACE, placed at the top of the directory tree, can ensure that a principal is granted or denied access to every object of a particular type.

```
Services: container type
       ACE1: inherited object type = null
       ACE2: inherited object type =
                                Web Service
       ACE3: inherited object type =
                                RPC Service
Services\WWW: Web Service type
       ACE1: inherited object type = null
       ACE2: inherited object type =
                                Web Service
Services\Names: RPC Service type
       ACE1: inherited object type = null
       ACE3: inherited object type =
                                RPC Service
```

Figure 7: ACLs on a container and two objects within the container. Type specific ACEs are inherited from the base ' Services' container only onto specific types of objects.

4.2 Static Inheritance

While type-specific inheritance ensures that different objects will inherit different ACLs, it does not ensure that changes to an ACL can be correctly propagated to a tree of objects. The primary difficulty is merging ACEs applied locally to an ACL with the entries inherited from its parent. Another difficulty is that inheritance must be limited, so that certain portions of a hierarchy, such as those containing private information, do not receive inherited access control.

Windows 2000 allows changes to access control lists to be propagated down a tree by annotating ACEs with inheritance information and by allowing inheritance to be disabled on an object or container. In addition, Windows 2000 specifies exactly how a locally modified ACL is merged with inherited access control entries. These changes allow inheritance rules to be reapplied with the same results, making the process of changing permissions on a hierarchy simpler and more predictable. In Windows 2000, it is possible to grant access to a tree of objects by granting access to the root of the tree and reapplying inheritance to the tree. If the propagation of inheritance aborts due to a failure, inheritance can be applied again with predictable results. This inheritance is *static*, because it 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 was dynamic and an object's ancestors were checked during an access check.

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. Another flag, the SE_DACL_PROTECTED flag on security descriptors, prevents ACEs from being inherited onto an ACL. This flag is stored on security descriptors, the container for security information on each object, rather than ACLs because a security descriptor with no access control list is interpreted as granting full access to all principals, and may need to be protected from inheritance.

In addition to adding these flags, the ordering rules for ACEs are different in 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, we chose to grant administrators of a sub-tree the ability to override all inherited permissions (which can also be accomplished by protecting the ACL from inheritance). Placing all access denied ACEs first prevents the administrator of an object from overriding an inherited ACE that denies access. As a result, in Windows 2000, locally added ACEs are placed first, followed by inherited entries. Unfortunately, this ordering limits centralized control by allowing high-level access control decisions to be overridden at lower levels in the hierarchy.

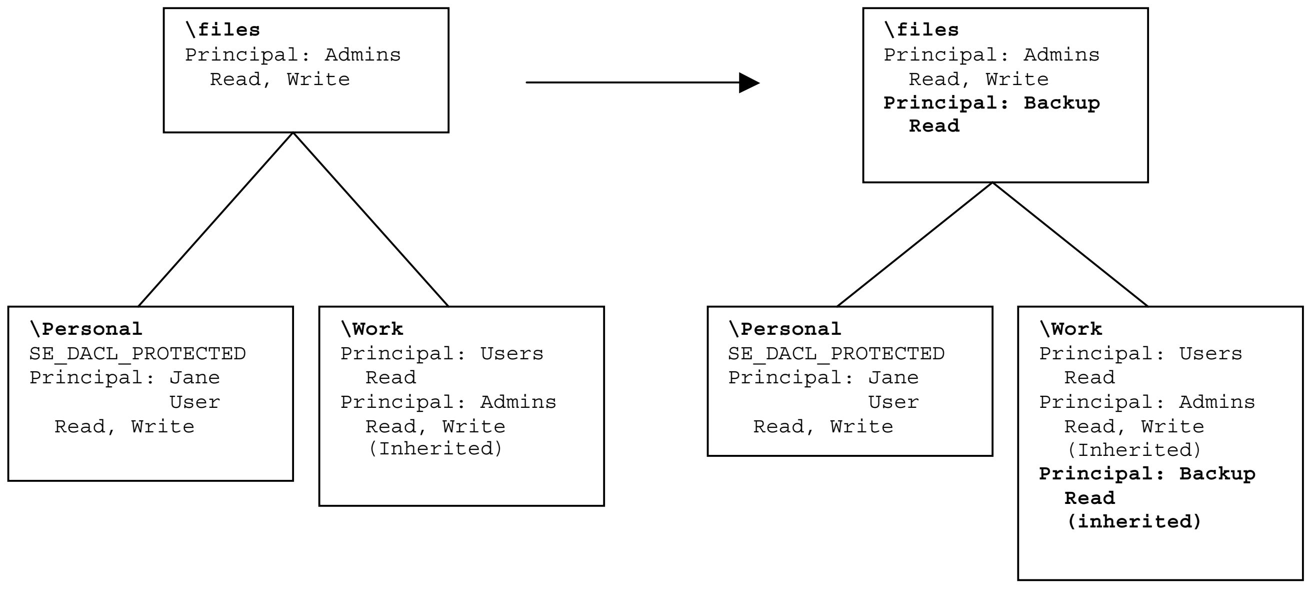

Figure 8: An example of reapplying inheritance. On the left is shown a directory hierarchy, in which the \*Personal* directory is protected from inheritance. After modifying the ACL on the \files directory with the bold-faced entries, the new ACE is propagated to the \Work directory but not the \Personal directory.

4.3 Example

An example of how these flags are used is shown in Figure 8. In this example, the '*\files\Personal*' directory overrides the inherited permissions by removing the access of administrators and instead granting access to the user '*Jane User*'. The other directory, '*\files\Work*', augments the inherited permissions by additionally granting the '*Users*' group read access. When a new ACE is added to the '*\files*' directory, the change is only propagated to the '*\files\Work*' directory, while the '*\files\Personal*' directory is protected from inheritance.

4.4 Discussion

These enhancements to the inheritance rules allow centralized management of access control because access permissions can be administered at any level of the hierarchy. Some accesses can be administered at the top and propagate down, while others are applied lower in the tree or directly to a single object. In addition, some portions of the tree may be more protected and block inheritance of rights from above. This approach provides the major benefit of dynamic access control, which is centralized administration high in the tree of objects. However, it 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 used mostly for making global changes to the Active Directory rather than to delegate access to particular object types. 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.

Compared to Windows NT 4.0, this inheritance mechanism increases the cost of access control both in space and time. First, 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, by only storing one copy of each unique ACL, the space needed to store ACLs can be reduced. Second, the larger ACLs make access check operations on containers more expensive [18], because every ACE, whether or not it is effective, 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. Another difficulty is that changing an object from a non-container to a container requires reapplying inheritance on every instance of the object, so that the inheritable ACEs can be propagated to its ACL. Finally, applying inheritance statically requires that some piece of code walk the tree of objects and re-apply inheritance. This re-application must be resumed if the machine crashes, and for certain applications, such as the Active Directory, must be transactional. The process is much more complex than dynamically applying inheritance during an access check. However, 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 lets administrators control which *users* may have access to an object, but it does not let users choose which *programs* may have access. The second major access control concern in designing Windows 2000 was limiting the damage caused by misbehaving programs. There are many examples of extra protection layered on top of operating system protection, such as Janus [10] and Tron [4]. These systems augment Unix protection with separate configuration files or scripts that limit the rights of certain programs and processes. 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

```
Access Token:
        User SID: Jane User

        Group SIDs: Everyone
                    GradStudents
                    SecurityGroup
                      (use for deny only)

        Restricted SIDs: StockTicker

        Privileges: none
```

Figure 9: A restricted token. In this token, the *Security-Group* group SID has been disabled so it can only be used to deny access. In addition, the *StocketTicker* SID has been added to the Restricted SIDs field, so that SID must be granted access to any objects accessed by this token.

```
ACL 1:
  Principal: Jane User
    Access: Read, Write, Delete
  Principal: StockTicker
    Access: Read
Access granted to restricted context:  read

ACL 2:
  Principal: Jane User
    Access: Read, Write, Delete
Access granted to restricted context:  none

ACL 3:
  Principal: SecurityGroup
    Access: Read, Write
  Principal: StockTicker
    Access: Read, Write
Access granted to restricted context:  none
```

Figure 10: This example shows the results of using the restricted token from Figure 9 to access objects protected by three ACLs.

one set of mechanisms rather than using different mechanisms for different access control purposes. Finally, multiple access control mechanisms protecting the same objects may cause confusing results or interact poorly.

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

- Code should have no greater access than the user running the code.
- Users should be able to restrict programs to accessing specific objects or classes of objects.
- No separate security model should be needed for restricting code.

These goals suggest that untrusted code should run in a separate process and address space with its own access token, and access control on objects should limit the process to a subset of the objects accessible to the user.

5.1 Restricted Contexts

Windows 2000 allows users to create a limited version of their identity that may access only a subset of the objects normally accessible. A *restricted context* is an access token with a *restriction*, which consists of a set of groups to disable, a set of privileges to disable, and a set of restricted security identifiers. The disabled privileges and groups cannot be re-enabled by the untrusted code. The groups are marked USE_FOR_DENY_ONLY, so they will be used with ACEs denying access but not those granting access. The restricted SIDs represent the identity and privileges of the program being run and are used during access checks. The user may choose these SIDs and assign SIDs to different programs or software publishers. 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. A user can grant a particular program access to an object by, first, creating a restricted SID for the program, then setting appropriate ACLs to grant access to that SID, and finally, running the program in a restricted context with that SID in the restrictions. As a result, users can choose with objects may be accessed by a program.

An example of a restricted context is shown in Figure 9, and how it is used for an access check is shown in Figure 10. In this example, the token has a single restricted SID, '*StockTicker*', representing a downloadable stock-ticker application. The three ACLs shown in Figure 10 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, the restricted context has no access, because there is no ACE granting '*StockTicker*' any access. Similarly, the third access is denied because the '*SecurityGroup*' SID in the unrestricted portion of the token may only be used to deny but not to grant access. As a result, the restricted context is granted access to only a subset of the resources available to the user.

Restricted contexts can be used to implement the policy of *least privilege* [23], which states that a program should have only the privileges necessary to perform its job and no more. As a result, some common application exploits, such as macro viruses, can be prevented because the application has no access to unrelated or unnecessary resources.

5.2 Applying Restrictions to System Objects

Implementing restrictions by placing restricted SIDs in access control lists poses three problems.

- Users do not control many objects in the operating system, so they cannot grant access to those objects.
- The operating system creates the ACL on many non-persistent objects, such as for user interface objects, so users do not have an opportunity to store a new ACL on these objects.
- The access rights for an operating system object may be at the wrong granularity, such as in the case of sockets, which do not distinguish between different endpoints.

Our solution to protecting system objects was to create 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

SID "restricted-network" is used to grant access to network components, and the "restricted-windows" SID grants access to the user interface. On resources for which ACLs do not provide the correct granularity of protection, access to the object itself must be denied. Instead, the untrusted application is linked to proxy code that contacts a trusted service to perform the operation. For example, network client code contacts a trusted proxy service to create a network connection. The proxy then checks whether the untrusted code is allowed to contact a specific endpoint and creates the connection. The proxy code may slow down the operation significantly if new connections are frequent, but is still effective at enforcing finer granularities of access control. These features were implemented but dropped for the final release of Windows 2000 due to schedule pressure.

5.3 Remote Authentication with Restricted Contexts

Restricted contexts are most useful for local access control but they may be extended across a network. If restricted contexts have direct access to a user's password, private keys for TLS [9], or Kerberos ticket cache [11], then the untrusted code can authenticate itself as the user to another machine without the restrictions by implementing the authentication protocol itself. However, by exposing only abstract authentication operations, such as the GSS-API [14], the operating system can ensure that the authentication protocol carries restrictions to remote servers. As long as the untrusted code has no opportunity to corrupt the restrictions, this technique allows restrictions to be used across machine boundaries. The Kerberos protocol used in Windows 2000 [11,12] has a field, *authorization-data*, in its encrypted authentication messages with which the client can explicitly limit its authority on the server. As a result, the Kerberos client code can bundle the restrictions from an access token and securely transmit them to a remote server, which can then create a restricted context for the client. This feature was also implemented but not shipped with Windows 2000.

5.4 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 allow users to divide their identity, 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. Similarly, users can choose to 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 usefulness 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 normal applications from exploitation by malicious viruses.

There are several issues with restricted contexts we have still not resolved. The first is assigning restricted SIDs to executables. Currently, we use the DNS domain name of the source of the executable, but this requires trusting DNS to accurately translate that name. The second issue is determining the correct context for a process executing code from multiple sources. Intersecting the two contexts may create too restrictive a context, and using the union of the two contexts may be too relaxed to be safe. Finally, restricted contexts do not fully isolate programs; they have no notion of information flow [3,8]. A rogue mail program in a restricted context may save a file that is later accessed from an unrestricted context where it may cause damage, so some form of labeling objects with their source is necessary to fully isolate programs.

6 RELATED WORK

The work presented here is an evolutionary step from the access control model in Windows NT 4.0. The inheritance model in Novell NDS 8 [21] resembles the model in Windows 2000, by supporting inherited rights in a directory service. NDS differs in that it uses dynamic inheritance. In addition, NDS supports inheritance *filters*, which are similar to our SE_DACL_PROTECTED flag except that they protect the inheritance of specific rights rather than all rights. Furthermore, NDS does not support grouping properties into property sets. WebDAV [7] also has a model similar to Windows 2000. It supports protecting objects from inheritance and marking ACEs as inherited, but does not restrict objects to a tree structure.

Restricted contexts are similar to the compound principals from [1], where two principals can be required for access. However, restricted contexts change the subject of access control rather than changing ACLs to require compound principals. Role-based access control (RBAC) [24] is related to restricted contexts in that programs can be run with different identities according to their task. Unlike RBAC, users can choose whether to use restricted contexts and what context to use.

Several projects provide isolation between processes to limit the exploitation of trusted code. HP's Compartmented Mode Workstaion (CMW) [27] and domain and type enforcement (DTE) in Unix [26] provide isolation between processes and allow for restricting the objects accessible to a process. However, while these systems provide stronger security guarantees, they also require far more changes to the operating system, such as support for mandatory access control, than do restricted contexts. The WindowBox project [2] provides isolation with restricted contexts by separating applications onto separate user-visible desktops rather than just running them in different contexts.

Janus [10] and Tron [4] are similar to our work, in that they protect the boundary between processes and the operating system by layering protection on top of operating system security mechanisms. Both approaches use a separate set of security policies and configuration tools for protecting users from untrusted code, and as a result are not integrated with the existing operating system security mechanisms. The benefit of these systems is that they provide more flexible policies, such as modifying the arguments to system calls.

Finally, [16] suggests 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. While many of the ideas have been seen before in other systems or in slightly different forms, their combination provides the right balance of feasibility, performance, and manageability needed for Windows 2000. In particular, extending access control entries to specify both to which portion of an object they refer and onto which objects they may be inherited 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 misbehaving code by allowing users to restrict the set of objects accessible to a program and provides the support needed to implement the policy of least privilege. Overall, these changes greatly improve the scalability and security of the Windows 2000 operating system.

8 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. David Ely provided valuable feedback on an early draft of this paper. We would also like to thank our anonymous reviewers and Robert Grimm, our shepherd, for his insightful and detailed comments that greatly improved the paper.

9 REFERENCES

[1] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin, *A Calculus for Access Control in Distributed Systems*. ACM Transactions on Programming Languages and Systems, 15(4):706--734, Oct. 1993.

[2] D. Balfanz, and D. Simon, *WindowBox: A Simple Security Model for the Connected Desktop*. In Proceedings of the 4th USENIX Windows Systems Symposium, Aug. 2000.

[3] D. Bell and L. LaPadula, *Secure Computer System: Unified Exposition and the Multics Interpretation*. Technical Report No. ESD-TR-75-306, Electronics Systems Division,AFSC, Manscom AF Base, Bedford, MA, 1976.

[4] A. Berman, V. Bourassa, and E. Selberg, *TRON: Process-specific file protection for the UNIX operating system*. In Proceedings of the 1995 USENIX Winter Technical Conference, pages 165—175. Jan. 1995.

[5] B. Callaghan, B. Pawloski and P. Staubach, *NFS Version 3 Protocol Specification*. Request for Comments RFC 1813, Internet Engineering Task Force, Jun. 1995.

[6] Computer Emergency Response Team, *CERT Advisory CA-2000-16 Microsoft 'IE Script'/Access/OBJECT Tag Vulnerability*. http://www.cert.org/advisories/CA-2000-16.html, Aug. 2000.

[7] G. Clemm, A. Hopkins, E. Sedlar and J. Whitehead, *WebDAV Access Control Protocol*. Internet draft draft-ietf-webdav-acl-04, Intnernet Engineering Task Force, Jan. 2001.

[8] D. Denning, *A Lattice Model of Secure Information Flow*. Communications of the ACM, 19(5), pages 236-243, Aug. 1976.

[9] T. Dierks and C. Allen, *The TLS Protocol*. Request for Comments RFC 2246, Internet Engineering Task Force, Jan. 1999.

[10] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. *A Secure Environment for Untrusted Helper Applications --- Confining the Wily Hacker*. In Proceedings of the 1996 USENIX Security Symposium.

[11] J. Kohl and B. C. Neuman. *The Kerberos Network Authentication Service (V5)*. Request for Comments (Proposed Standard) RFC 1510, Internet Engineering Task Force, Sep. 1993.

[12] J. Kohl, B. C. Neuman, and T. Y. T'so. *The Evolution of the Kerberos Authentication System*. In Distributed Open Systems, pages 78-94. IEEE Computer Society Press, 1994

[13] P. J. Leach and R. Salz, *UUIDs and GUIDs*. Internet Draft draft-leach-uuids-guids-01.txt. Internet Engineering Task Force, Feb. 1998.

[14] J. Linn, *Generic Security Service API*, Request For Comments RFC 1508, Internet Engineering Task Force, Sep. 1993.

[15] D. Mackey and R. Salz, *DCE ACL Library – Functional Specification*, OSF DCE SIG Request For Comments 46.0, Oct. 1993.

[16] D. Mazières and M. F. Kaashoek, *Secure Applications Need Flexible Operating Systems*. In Proceedings of the 6th Workshop on Hot Topics in Operating Systems, May 1997.

[17] Microsoft Corp., *Windows 2000 Active Directory*, http://www.microsoft.com/widows2000/guide/server/features/directory.asp, 2000.

[18] Microsoft Knowledge Base, *Large Numbers of ACEs in ACLs Impair Directory Service Performance*, http://support.microsoft.com/support/kb/articles/q271/8/76.asp, 2000.

[19] Microsoft Corp., *Distributed Component Object Model*. http://www.microsoft.com/com/tech/dcom.asp, 1998.

[20] Microsoft Corp., *ActiveX Controls*, http://microsoft.com/com/tech/activex.asp, 1999.

[21] Novell Inc., *NDS 8*. http://www.novell.com/documentation/lg/nds8/docui/index.html, 1999.

[22] D. Ritchie, and K. Thompson, *The UNIX Timesharing System*. Communications of the ACM, 17(7), pages 365-375, Jul. 1974.

[23] J. Saltzer and M. Schroeder. *The Protection of Information in Computer Systems*. In Proceedings of the IEEE 63(9), pages 1278-1308, Sep. 1975.

[24] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. *Role-Based Access Control Models*. IEEE Computer, 29(2) pages 38- 47, Feb. 1996.

[25] M. Swift, J. Trostle, J. Brezak and B. Gossman, *Kerberos Set/Change Password: Version 2*, Internet Draft draft-ietf-cat-kerberos-set-passwd-03 Internet Engineering Task Force, Apr. 2000.

[26] K. Walker, D. Sterne, M. Badger, M. Petkac, D. Shermann, and K. Oostendorp, *Confining Root Programs with Domain and Type Enforcement (DTE)*. In Proceedings of the 6th USENIX Security Symposium, Jul. 1996.

[27] Q. Zhong, *Providing Secure Environments for Untrusted Network Applications*. In Proceedings of the 2nd IEEE International Workshop on Enterprise Security, Jun. 1997.