MASARYKOVA UNIVERZITA
FAKULTA INFORMATIKY

Access control in operating systems

BACHELOR THESIS

Matej Csányi

Brno, 2006

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Advisor: Ing. Mgr. Zdeněk Říha, Ph.D.

Thanks

I would like to thank my advisor for his valuable suggestions and help.

Abstract

The thesis focuses on describing mechanisms that provide access control in different operating systems. We are not particularly interested in formal specification but in the concept, possibilities and the actual implementation of access control models. The range of operating systems (and file systems) is wide enough to provide a complex picture of how developers strive to provide means to protect our data (from unauthorized access). The thesis discusses the most common operating systems that implement discretionary access control.

Keywords

operating system, file system, permissions, auditing, ownership, discretionary access control

Contents

Introduction

Security is a very important feature of modern operating systems. However security in operating systems is way too wide topic for a bachelor thesis. That's why this work focuses on describing an important part of security in operating systems, which is access control. We discuss ways how access control works and means by which it is achieved. Lately even formal security models are applied in design of operating systems. However the object of this thesis is to provide a lookthrough of what is already available in some commonly used operating systems. Access control doesn't only provide access to files (or deny access), but it assures that only certain users are permitted to do certain operations in the system - send signal to process, access objects, etc.

Throughout the whole text, we will assume, that the operating system knows who the user using the operating system is. In other words, we do not care about authentication. The user entered the necessary information (usually user name and password) and logged into the operating system that is in a consistent state. Although authentication and authorization are very closely related, we will use common sense to distinguish them and focus on authorization.

The thesis describes how authorization mechanism is applied to subjects and objects. Subject of operating systems are (active) entities that communicate with the system and use its resources. The best example for a subject is the user or a process. Objects on the other hand are entities of the operating system that are accessed (requested) by the subject. The access control mechanism should ensure that subjects gain access to objects only if they are authorized to. Objects are usually represented by descriptors. A descriptor is a unique identifier for a object in a operating system, such as a file descriptor or a security descriptor. The process of authorization then consists of comparing the security context of the object and subject.

Access control has to be incorporated nearly into every part of a modern operating system. The most visible and used part of access control from the user point of view is file management. Although the operating system acts following access control policy in every area (memory management, process management), the user wants to protect his resources even when away from computer. Naturally, the protection of all areas excluding file management is an elementary requirement which doesn't require an extra effort from the user. However protecting data on a permanent storage from unauthorized access is partially the concern of the user - as far as access rights go. However setting rights does not include only data on some per-

manent storage but also resources that other subjects are allowed to use, i.e. printer, scanner, etc. To set permissions on objects in a simple and effective manner, the user-group-others approach can be applied and/or access control lists can be used.

Depending on areas of usage, there are three types of access control used:

- *Mandatory Access Control*. It is a multilevel secure access control mechanism. It defines a hierarchy of levels of security. A security policy defines rules by which the access is controled. Used by the Department of Defense.

- *Discretionary Access Control*. In this type of access control, every object has an owner. The owner (subject) grants access to his resources (objects) for other users and/or groups. This way we can represent the rights of users with the following matrix: The matrix defines the

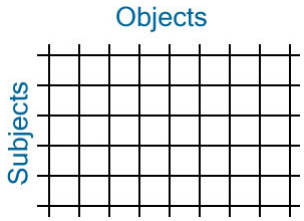

Figure 1: Discretionary access control matrix

  whole state of the system concerning the rights of individual users. There are two ways how to implement the matrix. Either the system assigns the rights to the objects or to the subjects. In other words, either the object stores the column of the matrix, or the subject stores the row of the matrix. Access control lists are used to store the rights with object. On the other hand capability matrixes are used to store rights together with subjects. In the case of capability matrixes we would have to deal with biometrics, so in common operating systems access control lists are used to implement discretionary access control.

- *Role Based Access Control*. In some environments, it is problematical to determine who the owner of resources is. In role based systems, users get assigned roles based on their functions in that system. These systems are centrally administered, they are nondiscretionary. An example is a hospital.

The whole thesis deals with operating systems that provide discretionary access control (if any). The concept of the object storing rights together with itself is very natural. This way, the rights are a property of the object. We can see this reflecting on some filesystems. Once again, during the process of authorization, the the ACL of a given object (file, directory, etc.) determines whether the subject has the rights to access the it.

Chapter 1

Microsoft

1.1 MS-DOS

DOS (Disk Operating System) is a 16bit singletasking OS operating under real mode. Being a single-user operating system, basically MS-DOS does not have any kind of authentication or authorization mechanisms (except for the file attributes that are stored on FAT partition(s)). Up to version 2.00 there was no support for hard discs, there could be only one directory (the root directory) and no further subdirectories could be created. MS-DOS consists of three main files which are needed for it to run:

- *IO.SYS*. Contains device drivers
- *MSDOS.SYS*. Provide file management, memory management, etc.
- *COMMAND.COM*. Implements the shell and basic commands

After newer processors appeared, such as 80386, the amount of addressable memory has suddenly grown to 4GB. MS-DOS has not been modified to support more than 1MB of memory, but two different mechanism were developed for MS-DOS to be able to use more than 1MB. These were:

- *Expanded memory* made MS-DOS support 16MB of memory
- *Extended memory* enabled to address memory up to 16MB for a 286 processor and for a 386 (or newer) processor up to 4GB.

1.2 Windows 1.x, 2.x, 3.x

Windows 1.x was Microsoft's first operating system with GUI. Also support for multitasking and virtual memory was added. However, the multitasking capability was limited and virtual memory system was software-based. In the terms of access control, Windows 1.x is identical to MS-DOS.

A specific version of Windows 2.x - Windows/386 - made possible to run multiple applications simultaneously in the extended memory.

As for Windows 3.x, worth mentioning is Windows for Workgroups 3.1, which brought (for the first time) networking support into a Windows. That way, file and printer sharing became available, and some access control technique could be applied to set the permissions of files and/or printers shared trough the network.

1.3 Windows 9x

All of the Windows 9x (Windows 95, Windows 98 and Windows Millennium) operating systems are still single-user, are mixed of 16bit and 32bit code and support preemptive multitasking. As of stability, [5]Windows 95, Windows 98, and Windows Millennium Edition have some pages in system address space that are writable from user mode, thus allowing an errant application to corrupt key system data structures and crash the system. Windows 95 uses VFAT file system and thus supports long file names (discussed in Filesystems chapter). Starting Windows 95 OSR2 upwards, Windows supports FAT32 file system also.

On Windows 9x operating systems two different approaches can be used when controling access to shared resources.

1.3.1 Share-level access control

Share level access provides a simple method for sharing resources. This sharing than applies to all users that want to access the shared resource. We can either share the resources for "Full Access" or "Read-Only Access". Full access means that files and directories are allowed to be read, written and deleted. We can also specify a password needed to access the resource in "Full Access" and/or "Read-Only Access". The advantage of this security paradigm is that it allows granting access to a large number of people with very little effort. However, it is not very secure, since the password is widely distributed and there is no notion of personal accountability. [2]

1.3.2 User-level access control

Pass-through user-level security protects shared network resources by requiring that a security provider authenticate a user's request to access resources. The security provider, such as a Windows NT domain control or NetWare server, grants access to the shared resource by verifying that the user name and password are the same as those on the user account list stored on the network security provider. Because the security provider maintains a network-wide list of user accounts and passwords, each computer running Windows 95 does not have to store a list of accounts. [3] By using this kind of access control, we gain much finer control over who may access our data and in what way. Beside the already mentioned "Full Access" and "Read-only Access", we can select and combine from a variety of different access rights. The possible rights are: Read Files, Write To Files, Create Files and Folders, Delete Files, Change File Attributes, List Files, Change Access Control. This paradigm allows fine-grained control over per-user access and allows individual accountability. The disadvantage is that you must create a user account for each user you want to grant access to and you must grant that user the access (either directly or by adding the user to an appropriate group). [3]

1.4 Windows NT 4

Windows NT was the Microsoft's first operating system with huge emphasis on security. It is a full 32bit operating system supporting preemptive multitasking. Emphasis is not only placed on providing a wide variety of services, but also monitoring and auditing parts of the system. Under Windows NT we are able to view how many sessions, open files, file locks and open named pipes currently are in the system. We can view which resources are shared, and which of these resources are being used by users.

1.4.1 Directory Replication

Directory replication is a way of synchronising directories on a network. We can either export or import a directory. In case of exporting, a directory to be exported is selected and then domain(s) and/or computer(s) are added to the replication to list. The computer(s) in the replication to list (or computer(s) in a domain that is in the list) specify an import directory and add the computer that is exporting a directory to their replication from list. Due to the directory replication, as soon as any changes are made in the directory that is exported, these changes will reflect on every computer that imports this directory.

1.4.2 Account Policy

Account policy enables us to set some security "restraints" on accounts. We can set password restrictions:

- *Maximum password age*. Can be set to "Password never expires" or to "Expires in X days".
- *Minimum password age*. Can be set to "Allow changes Immediately" or "Allow changes in X days".
- *Minimum password length*. We can either "Permit blank password" or require "At least X characters".
- *Password uniqueness*. We can assure that a newly selected password is unique by remembering the last few passwords. Or we can simply choose "Do not keep password history" to allow the user to use the same password.

The *Account lockout* option enables us under a certain condition to automatically lockout an account. The condition is that there has been a certain amount of bad logon attempts. The amount of minutes when a bad logon attempt counter gets reseted may be set also. If a account gets locked out, we can select the lockout duration to forever (until the admin unlocks the account) or to a certain amount of minutes. We can also tell the system to forcibly disconnect remote users from server when logon hours expire. *Logon hours* can be specified, so users will be allowed to logon in certain hours.

1.4.3 User Rights Policy

User rights policy determines which groups of users and/or individual users are to be granted certain kinds of rights in an Windows NT enviroment. We will only enumerate the possible rights as they are self-explanatory. They are: Access this computer from network, Add workstations to domain, Back up files and directories, Change the system time, Force shutdown from a remote system, Load and unload device drivers, Log on locally, Manage auditing and security log, Restore files and directories, Shut down the system, Take ownership of files or other objects. We can also choose from advanced user rights, which are: Act as part of the operating system, Bypass traverse checking, Create a pagefile, Create a token object, Create permanent shared objects, Debug programs, Generate security audits, Increase quotas, Increase scheduling priority, Lock pages in memory, Log on as a batch job, Log on as a service, Modify firmware environment values, Profile single process, Profile system performance, Replace a process level token.

1.4.4 Audit Policy

In Windows NT we can select to audit some of the most common events in the system. We can select to audit the *success* and/or the *failure* of the following events:

- *Logon and logoff*
- *File and object access*
- *Use of user rights*
- *User and group management*
- *Security policy changes*
- *Restart, shutdown and system*
- *Process tracking*

1.4.5 Per File (Directory) Security

In Windows NT we can control the security features on a per-file basis. Naturally security properties can be set on directories, but Windows NT allows to, for instance, share two files under one directory differently. There are three security properties that can be set: permissions, auditing and ownership.

Permissions

We can define permissions that will be applied to groups of users and/or to individual users. There are six permissions available:

- *Read (R)*. User can enter the directory, list its contents and read the contents of a file.
- *Write (W)*. Allows to write data in directory - create new files, although cannot delete files.
- *Execute (X)*. User can execute files (if they are executable).
- *Delete (D)*. Allows to delete file or directory.
- *Change Permissions (P)*. Allows to change permissions of a file or directory.
- *Take Ownership (O)*. User can take ownership of file or directory.

However Windows NT combines the above mentioned permissions into seven more general types of access for directories which make assigning security properties a little easier and more intuitive.

- *No Access*. User can't do anything with the directory and files in it.
- *List (RX)*. User can enter directory and list files, but cannot read or modify them.
- *Read (RX)*. Applies to the directory and the files in it also. Same as list, but user is allowed to read and execute files.
- *Add (WX)*. Allows to enter directory and store files in it. However files cannot be read, changed or deleted.
- *Add & Read*. (RWX) for the directory and RX permissions for the files in it. Same as "add", but user is allowed to read and execute files in the directory.
- *Change*. Combination of RWXD permissions for the directory and the files in it also. User is allowed to enter directory, list the contents, add, change, execute and delete files in it.
- *Full Control*. User gets every permission including the right to change permissions or take ownership of files and directories.

While assigning permission for directories, we can select whether we want to replace permissions on subdirectories and whether we want to replace permissions on individual files.

When setting the permissions of individual files we have four predefined types of access:

- *No Access*
- *Read (RX)*.
- *Change (RWXD)*

- *Full Control (all permissions)*

In case that none of the predefined types of access fit our needs (while asigning permissions to files or directories), using special file access or special directory access, we can select (and combine) from the ”basic” permissions.

Auditing

Windows NT allows to audit every event that ocurred on a file or directory. The events are the same as permissions. This is very logical, because we want to have control (define permissions) and overview (audit) of all possible events that can happen with a file or directory. We select to audit success of an event, failure of an event or both. While configuring auditing of a directory, we can select to replace auditing on subdirectories and also replacing auditing on existing files. Windows NT allows us to audit different events for different groups of users or even individual users.

1.4.6 Permission Resolution

Permission resolution determines whether a subject is allowed to access an object based on assigned permissions. This may seem trivial, as intuitively it is enough to check the ACLs of the object. However Windows NT uses permission inheritance, allows to set permissions on individual files explicitly and a user can be a member of more than one group. For instance, a user can be a member of two groups. The object the user wants to access defines read permissions for one group, however denies them for the other group. Does the user have permission to read the object ? Permission resolution addresses this issue by specifying two rules by which the real rights become evident. The rules are:

- Permissions that allow access add on. Let’s sat Mathew is a member of Students group and wants to write into a file. The file specifies only read permissions for the group Students, however specifies write permissions for the user Mathew. The permissions add on, Mathew has both read and write permissions on the file.
- ”No Access” is a special (in this context) kind of permissions that takes precedence before every other permissions. In our example it would mean, that if the file would have ”No Access” permission for the group Students, no matter what permission the file would define for Mathew, he couldn’t do anything with the file. This is why ”No Access” has to be used very carefully, because it can completely take away any rights of users (groups).

1.4.7 Static Inheritance

Inheritance of permission makes it very easy to assign permissions for files and directories on big volumes with vast amounts of files and directories.

Directory trees can be however very deep and sometimes the Windows NT static inheritance can cause troubles.

When an file or directory is created, its automatically given the permissions from its parent directory. In other words newly created files and directories inherit the permissions of their parent directory. By default this kind of inheritance is applied only once. If the parent directory's permissions get changed, his child's permissions remain the same. Windows NT provides the already mentioned options when setting directory permissions which are: "replace permissions on subdirectories" and "replace permissions on existing files". This way we can propagate permissions throughout the whole directory hierarchy. However, the administrator (or whoever is applying the permission inheritance) has to be very cautious. While the permissions are replaced on subdirectories and/or existing files, whatever permissions they had before were totally replaced by the new permissions. This way every customization of permissions in the directory tree are gone, and the same permissions are applied to all files and/or directories. The solution (under Windows NT) is to set and propagate permissions deeper in the directory tree and that way avoid the "disappearance" of set permissions. Windows 2000 uses dynamic inheritance which addresses these issues. By installing service pack 4 in Windows NT we also gain dynamic inheritance.

1.5 Windows 2000

Windows 2000 is a very popular 32bit operating system based on the Windows NT 4 kernel. It implements a POSIX subsystem (based on the POSIX.1 standard) and also an OS/2 subsystem. We will describe Windows 2000 in a little more detail as it is a modern operating system with very sophisticated security features and architecture. Windows 2000 has three versions:

- *Server* can handle 4 processors and 4GB of memory.
- *Advanced Server* supports up to 8 processors and 8GB of memory. Also supports two node clustering.
- *DataCenter Server* supports up to 32processors and 64GB of memory. Also supports four node clustering.

Every Windows 2000 process comprises a security context called an access token that identifies the user, security groups, and privileges associated with the process. By default, threads don't have their own access token, but they can obtain one, thus allowing individual threads to impersonate the security context of another process including processes running on a remote Windows 2000 system without affecting other threads in the process. [5] The impersonating feature can be compared to effective IDs in Unix-like operating systems.

Each process has data structures associated with it. On of them is the process block which contains (among other) the process id, the id of the parent process, quota block, process priority class, pointer to the primary access token and a pointer to handle table. Access token object describes the security profile of the process. Threads have also a data structure called thread block, which is very similar to a process block. It contains (among others) the process ID, impersonating information and a pointer to the access token.

Windows 2000 contains components that control and insure security. Some of them are the following:

- *Security reference monitor* enforces security policies on the local computer. It guards operating system resources, performing runtime object protection and auditing.
- *Object manager* creates, manages, and deletes Windows 2000 executive objects and abstract data types that are used to represent operating system resources such as processes, threads, and the various synchronization objects.
- *Local security authentication server process* actually performs authentication requests. For instance, after the authentication, it creates an access token object representing the user's security profile.

When a process wants to access an object, it creates a handle to an object. Next the object manager sends the security reference monitor the set of the process's access rights. The object's security descriptor is checked by the security reference monitor to determine whether the type of access the process is requesting is permitted. If permitted, the security reference monitor gives a set of granted access rights to the process and the object manager stores them in the object handle.

In Windows 2000 shared memory sections have access control lists that determine whether a process will gain access to them.

1.5.1 Resource Accounting

Windows 2000 object manager provides resource accounting. Each object header contains an attribute called quota charges that records how much the object manager subtracts from a process's assigned paged and/or nonpaged pool quota when a thread in the process opens a handle to the object. Each process on Windows 2000 points to an already mentioned quota structure that records the limits and current values for nonpaged pool, paged pool, and page file usage. All the processes in an interactive session share the same quota block. System processes, such as services, have no quota limits. Windows 2000 doesn't enforce quotas. The default quotas of paged and nonpaged pool for a process are set to 0, which means no limit. We can change the default values, however the limits are soft. The system tries to

increase the process quotas when they are exceeded. The increase of quotas is done by the memory manager.

1.5.2 Memory Manager

Windows 2000 allows to set options concerning memory related settings. *ClearPageFileAtShutdown* specifies whether inactive pages in the paging file are filled with zeros when the system is shut down. This is a security feature.
IoPageLockLimit specifies the limit of the number of bytes that can be locked in a user process for I/O operations. When this value is 0, the system uses the default (512 KB).
NonPagedPoolQuota indicates the maximum nonpaged pool that can be allocated by any process (in megabytes). If the value of this entry is 0, the system calculates the value.
NonPagedPoolSize indicates the initial size of nonpaged pool in bytes. When this value is 0, the system calculates the value.
PagedPoolQuota indicates the maximum paged pool (in megabytes) that any process can allocate. If the value of this entry is 0, the system calculates the value.
PagedPoolSize indicates the maximum size of paged pool in bytes. When the value of this entry is 0, the system calculates the value. A value of -1 indicates that the largest possible size is selected, which means allowing a larger paged pool in favor of system page table entries (PTEs).

1.5.3 Security Identifiers and Access Tokens

Each subject that can perform some kind of action in Windows 2000 has a security identifier (SID) assigned. A SID is a variable-length numeric value that consists of a SID structure revision number, a 48-bit identifier authority value, and a variable number of 32-bit subauthority or relative identifier (RID) values. SIDs, being long and generated out of random sequences, it is almost impossible for two SIDs to be equal. Each process or thread has an access token that represents it's security context. A security context consists of information that describes the privileges, accounts, and groups associated with the process or thread. During the logon process a access token is created and attached to the user's logon shell process. Every program that afterward the user executes will inherit a copy of the initial token.

There are two components in a token that determine what a thread or process is privileged to do:

- *SID a group SID(s)* are the user account security ID and group security ID(s). SID is used by the security reference manager to determine whether a process or thread can gain access to an object. The SID(s) represent groups of which the user is member of.

- *Privilege array* is a list of rights that determines what a thread or process can do.

A token has a default primary group and default ACL associated with it, which is inherited by objects created by the process.

1.5.4 Security Descriptors and Access Control

Tokens determine what a user or process can do. Security descriptors on the other hand determine who can perform what actions on objects. Security descriptors composed of:

- *Revision number* determines the version of the security reference manager security model.
- *Flags* define the behavior or characteristics of the descriptor.
- *SID* represents the owner's security ID.
- *Group SID* represents the primary group's security ID. This field is used only by POSIX.
- *Discretionary access control list* determines who can gain access to the object.
- *System access control list* determines the auditing for various user or groups of users.

Access control lists contain a header and a number of access control entries (ACE). There are two types of ACLS:

- *Discretionary Access Control Lists* contain a list of permissions that are granted or denied for certain users or groups of users. The discretionary access control entries contain the ACE type, flags and ACE size. There are four types of ACEs:
  - *Access allowed* means that access is allowed.
  - *Access denied* means that access is denied.
  - *Allowed-object* specify permissions that only apply in Active Directory and allow access to some object.
  - *Denied-object* denies access to some objects within Active Directory.

  The *flags* in the ACE address inheritance issues. The possible flags are the following: [7]

  - *CONTAINER_INHERIT_ACE* Child objects that are containers, such as directories, inherit the ACE as an effective ACE. The inherited ACE is inheritable unless the NO_PROPAGATE_INHERIT_ACE bit flag is also set.

- *FAILED_ACCESS_ACE_FLAG* Used with system-audit ACEs in a system access control list (SACL) to generate audit messages for failed access attempts.
- *INHERIT_ONLY_ACE* Indicates an inherit-only ACE, which does not control access to the object to which it is attached. If this flag is not set, the ACE is an effective ACE which controls access to the object to which it is attached. Both effective and inherit-only ACEs can be inherited depending on the state of the other inheritance flags.
- *INHERITED_ACE* Indicates that the ACE was inherited. The system sets this bit when it propagates an inherited ACE to a child object.
- *NO_PROPAGATE_INHERIT_ACE* If the ACE is inherited by a child object, the system clears the OBJECT_INHERIT_ACE and CONTAINER_INHERIT_ACE flags in the inherited ACE. This prevents the ACE from being inherited by subsequent generations of objects.
- *OBJECT_INHERIT_ACE* Noncontainer child objects inherit the ACE as an effective ACE. For child objects that are containers, the ACE is inherited as an inherit-only ACE unless the NO_PROPAGATE_INHERIT_ACE bit flag is also set.
- *SUCCESSFUL_ACCESS_ACE_FLAG* Used with system-audit ACEs in a SACL to generate audit messages for successful access attempts.

The *ACE size* specifies the size of the ACE in bytes. Important to realize is that if no DACL is present (a null DACL) in a security descriptor, everyone gains full access to the object. On the other hand if the DACL is empty (contains zero ACEs), nobody has any kind of access to the object.

- *System Access Control List* contains two types of ACEs, system audit ACEs and system audit-object ACEs. The owner of the object has no control over SACLs. He cannot even read them. ACEs in SACLs specify which operating performed on the object by certain users or groups of users should be audited. Operations can be audited both on success and failure. The audit-object ACEs contain a GUID that specifies objects that should be audited. If there is no SACL present (SACL is null), no auditing takes place on the object. The inheritance flags that apply to DACL ACEs also apply to system audit and system audit-object ACEs.

1.5.5 File and Directory Permissions

Windows 2000 defines 13 permissions for files and directories. And to provide even more choices we can decide whether the permission will *allow*

access or *deny* access.

- *Traverse Folder / Execute File*
- *List Folder / Read Data*
- *Read Attributes*
- *Read Extended Attributes*
- *Create Files / Write Data*
- *Create Folders / Append Data*
- *Write Attributes*
- *Write Extended Attributes*
- *Delete Subfolders and Files*
- *Delete*
- *Read Permissions*
- *Change Permissions*
- *Take Ownership*

And also like Windows NT, Windows 2000 predefines 6 (5 for files) ”basic” permissions which are more intuitive and should be sufficient for most common tasks.

- *Full Control*
- *Modify*
- *Read & Execute*
- *List Folder Contents* naturally can’t be set on files.
- *Read*
- *Write*

1.5.6 Permission Resolution

Permission resolution under Windows 2000 is slightly complicated, because it uses dynamic inheritance and this way permission directly set on files can conflict with permissions inherited from their parent. Also a user can be in more than one group and Windows 2000 also add ”allow” and ”deny” permissions. Permission resolution follows four rules:

- ”Deny” permission take precedence over ”Allow” permissions.

- Directly set permissions take precedence over inherited permissions.
- Permissions inherited from near predecessors take precedence over permissions inherited from more distant predecessors.
- Permission inherited from more groups add on (just like in Windows NT).

1.5.7 Auditing

The auditing is very similar to Windows NT, but it is combined with dynamic inheritance. Windows 2000 allows to audit either success or failure of access events, or both. Access events are the same as permissions and auditing can be set for groups and/or individual users. When setting auditing on directories we have the same options (concerning inheritance) like with permissions, only in terms of auditing. These options are described in the "Dynamic Inheritance" subsection.

1.5.8 Dynamic Inheritance

Windows 2000 improved inheritance control compared to Windows NT by providing dynamic inheritance and additional options concerning inheritance. Same as in Windows NT, when a file or directory is created it automatically inherits all permissions from its parent. However (unlike in Windows NT) the object remains linked with its parent and any permission changes on the parent are propagated through the directory tree to its successors. This propagation is not destructive, as it keeps all the permissions that were directly set on the successors. This is possible, because permissions directly set on objects are stored "separately" from inherited permissions. Even in very deep directory trees, the inherited permissions "build up" and thanks to "deny" permissions the accumulated permissions can be narrowed. Naturally it all depends on the specific situation, but assigning permissions even within very deep directory trees can be done easily.

Not only the permission inheritance is improved (compared to Windows NT), we can select additional options when assigning permissions:

- *Child Protection* allows an object the option to or not to inherit permissions from parents by setting the "Allow inheritable permissions from parent to propagate to this object" option.
- *Object Selection Control* allows to specify objects to which permissions should be applied. It is only available when setting permissions on a folder. The options are:
  - This folder only
  - This folder, subfolders and files
  - This folder and subfolders

  - This folder and files
  - Subfolders and files only
  - Subfolders only
  - Files only

- *Recursion Control*. By setting the option "Apply these permissions to objects and/or containers within this container only" we can specify to propagate permissions only to the directories immediate successors and not "further".

- *Forced Propagation* can be achieved by selecting the option "Reset permissions on all child objects and enable propagation of inheritable permissions" and basically means that every permission set on the directory will be propagated to all children in the whole directory tree. Also this will remove any permissions on the child objects that were previously set. This option allows to set the same permissions for directory trees.

Chapter 2

Unix

Typically Unix and Linux operating systems implement a basic three-level access control protection for files. Access rights can be defined to three classes:

- *User (u)* specifies the user, possibly the owner of the file or directory.
- *Group (g)* specifies the users that belong to the same group like the file.
- *Others (o)* represent all other users.

For each class we can select three kinds of access rights: *Read*, *Write* and *Execute*.
These permission bits are checked in the following order:

- If the UID of the file is the same as the UID of the process, only the owner permissions apply; the group and other permissions are not checked.
- If the UIDs do not match, but the GID of the file matches one of the GIDs of the process, only the group permissions apply; the owner and other permissions are not checked.
- Only if the UID and GIDs of the process fail to match those of the file are the permissions for all others checked. If these permissions do not allow the requested operation, it will fail.

Unix defines three additional attributes:

- *suid*. After executing a file, the created process usually has its User ID equal to the ID of the user who executed the program. However, if the suid bit is set on an executable, the created process gets the ID of the owner of the file.
- *sgid*. A process normally has the Group ID equal to the process group. If however the sgid bit is set on the executable, the process is assigned the Group ID of the file group.
- *sticky*. The sticky bit of a file requests for the kernel to keep the process in memory after termination.

Each file has an owner and a group. A UID of 0 specifies the superuser (root), while a GID of 0 specifies the root group. Only the owner and the root can change the permissions. The privilege of changing permissions of files cannot be shared. This basic access control mechanism is used in all Unix operating systems. Some of them however add more up-to-date features for access control in form of access control lists.

Chapter 3

Linux

3.1 2.6.x kernel

3.1.1 Process Credentials

In Unix operating systems each process has some *credentials*. The credentials determine what the process is and isn't allowed to do. Naturally the credentials of the process are only one side of the coin. There have to be objects being protected from processes. That is reached by using discretionary access control. The credentials of the process are stored in the process descriptor. The process descriptor contain fields that are compared with the access rights of the objects the process wants to access.

Name	*Description*
uid, gid	User and group real identifiers
euid, egid	User and group effective identifiers
fsuid, fsgid	User and group effective identifiers for file access
groups	Supplementary group identifiers
suid, sgid	User and group saved identifiers

Table 3.1: Traditional process credentials [11]

3.1.2 Process capabilities

Capabilities of a process represent fairly special operations that a process is either allowed to do or not. These capabilities are closing the huge gap between user rights and superuser rights. Some of the credentials are:

- *CAP_CHOWN* Ignore restrictions on file user and group ownership changes.
- *CAP_DAC_OVERRIDE* Ignore file access permissions.
- *CAP_DAC_READ_SEARCH* Ignore file/directory read and search permissions.
- *CAP_FOWNER* Generally ignore permission checks on file ownership.
- *CAP_FSETID* Ignore restrictions on setting the setuid and setgid flags for files.

- *CAP_KILL* Bypass permission checks when generating signals.
- *CAP_IPC_LOCK* Allow locking of pages and of shared memory segments.
- *CAP_LINUX_IMMUTABLE* Allow modification of append-only and immutable Ext2/Ext3 files.
- *CAP_NET_ADMIN* Allow general networking administration.
- *CAP_SETGID* Ignore restrictions on group's process credentials manipulations.
- *CAP_SETPCAP* Allow capability manipulations.
- *CAP_SYS_ADMIN* Allow general system administration.
- *CAP_SYS_NICE* Skip permission checks of the nice( ) and setpriority( ) system calls, and allow creation of real-time processes.
- *CAP_SYS_PACCT* Allow configuration of process accounting.
- *CAP_SYS_RESOURCE* Allow resource limits to be increased.

3.1.3 Process Resource Limits

Linux provides resource limits to control the amount of resources that a process can use. Each process has a set of resource limits associated with it. The resource limits are the following:

- *RLIMIT_AS* defines the maximum address space of the process in bytes.
- *RLIMIT_CORE* specifies the maximum size of a core dump in bytes.
- *RLIMIT_CPU* specifies the maximum amount of CPU time a process can execute in seconds.
- *RLIMIT_DATA* specifies the maximum heap size in bytes.
- *RLIMIT_FSIZE* determines the maximum file size.
- *RLIMIT_LOCKS* constrains the maximum number of file locks.
- *RLIMIT_MEMLOCK* determines the maximum amount of nonswappable memory in bytes.
- *RLIMIT_NOFILE* constrains the maximum number of open file descriptors.
- *RLIMIT_NPROC* specifies the maximum number of processes a user can own.

- *RLIMIT_RSS* determines the maximum number of page frames owned by the process.

- *RLIMIT_STACK* specifies the maximum stack size in bytes.

3.1.4 Access Control Lists

Linux uses POSIX ACLs to achieve better access control for files and directories. The traditional access control in Linux is the same as in Unix operating systems which we described in the beginning of the Unix chapter. The use of ACLs under Linux is definitely available with the following filesystems: ReiserFS, ext2, ext3, JFS and XFS. The presence of ACLs under Linux allows us to migrate "environments" from Windows operating systems to Linux and the other way around. *Access ACLs* determine the user and group access permissions for files and directories. *Default ACLs* can only be applied to directories. They determine the permissions a file object inherits from its parent directory when created. *ACL entry* contains a type, a qualifier for the user or group to which the entry refers, and a set of permissions. There are two types of ACLs based on ACL entries. The *minimum ACL* consists from entries that assign permissions to the owner, owning group, and others which correspond to the conventional permission bits of the object. The *extended ACL* must contain a *mask* entry and can contain entries for the *named user* and *named group* types. The following table contains the ACL entry types:

Type	*Text Form*
owner	user::rwx
named user	user:name:rwx
owning group	group::rwx
named group	group:name:rwx
mask	mask::rwx
other	other::rwx

Table 3.2: ACL Entry Types

The permissions concerning "owner" and "others" are always effective. Named user, owning group and named group entries can be either effective (if a mask contains permissions) or masked. If setting permissions to named user or named group entry, a mask entry is generated. mask specifies the maximum effective access permissions for all entries in the group class. This includes: owning group, named user and named group. The *default ACL* on a directory defines the access permissions for objects created within the directory. A file inherits the default ACL from its parent directory as its own access ACL. A directory inherits the default ACL of the parent directory both as its own default ACL and as an access ACL.

When a system call creates a object, it uses the *mode* parameter to set the permissions of the object. However there are two possible scenarios that can occur:

- If the parent directory doesn't have a default ACL, the permissions of the object are determined by the intersection of the mode parameter and the umask set permissions.
- If the parent directory has a default ACL, the permissions of the object are determined as the intersection of the value of the mode parameter and the permissions specified in the default ACL.

The permission resolution happens in the following order: owner, named user, owning group or named group, and the other. Permissions do *not* accumulate.

Chapter 4

BSD

From FreeBSD, the NetBSD, and the OpenBSD project we are going to focus only on FreeBSD because of space issues.

4.1 FreeBSD

Monolithic kernel

4.1.1 Resource Utilization

While a process executes, the kernel keeps track of its resource utilization. The following resources are tracked:

- *The amount of user and system time used by the process*
- *The memory utilization of the process*
- *The paging and disk I/O activity of the process*
- *Number of voluntary and involuntary context switches taken by the process*
- *The amount of interprocess communication done by the process*

4.1.2 Accounting

FreeBSD provides a very simple method to resource accounting. When a process terminates, the kernel writes the accounting information into a accounting file. The superuser initiates accounting by assigning a file to the kernel for accounting purposes. Accounting is suspended when the free space on the filesystem drops below 2%, but is reinitiated after the free space reaches 4%. The accounting information of a process contains the following:

- *The name of the command that ran*
- *The amount of user and system CPU time that was used*
- *The time the command started*
- *The elapsed time the command ran*
- *The average amount of memory used*

- *The number of disk I/O operations done*
- *The UID and GID of the process*
- *The terminal from which the process was started*
- *Flags determining whether the process used superuser privileges, ran in compatibility mode, dumped core, and/or was killed by a signal.*

This kind of accounting isn't very reliable because when the system crashes, obviously no accounting information is stored no matter how long certain processes have been runing.

4.1.3 Resource Limits

FreeBSD implements almost the same (with minor exceptions) resource limits as Linux which is described in Linux chapter. However FreeBSD provides the possibility to control the *soft limit* and *hard limit* for each resource. The soft limit can be changed by all users from range zero to the value of the hard limit. If a process exceeds the soft limit for some resources a signal will be sent to the process. The signal will probably cause the process to terminate. If not then if the process won't release the resources, further attempts to allocate resources will cause errors. The users can also lower (only lower !) the hard limit, but only the superuser will be able to raise it.

4.1.4 Jails

To provide a secure and isolated enviroment FreeBSD provides jails. A possible solution would be using chroot, however [12]even with stronger semantics, the chroot system call is insufficient to provide complete partitioning. Jails are built on top of chroot and provide an enviroment in which processes can do whatever they want - within the jail. Outside the jail they have denied access and even visibility to the files and processes. Each jail can manage its own local policy independently from the host enviroment. On the other hand the jail is no different then the host enviroment and that way it has all the functionality - programs, network services - at disposal. Two requirements have to be met when considering jails: the discretionary access control mechanism must retain and each jail is allowed to have one superuser who can administer only files and processes within the jail. Each jail is associated with an IP address. A process that is authorized to create a jail must specify a IP address and the filesystem. Any children of this process will be in the jail as they inherit the prison structure from their parent. A process may only be in one jail and it can never leave the jail. The host system in which jail(s) reside can place further restrictions on the jail. Processes in jail that do not run with superuser privileges won't notice that they are in a "restricted" enviroment. However processes runing with superuser privileges will have restrictions placed on them when making system calls that could affect the host enviroment.

Chapter 5

Apple

5.1 Mac OS X Tiger

Mac OS X version Tiger is a UNIX-based operating system used on Apple computers. However support for the x86 platform has been added. Tiger is based on FreeBSD 5 and uses a microkernel architecture Mach 3.0. Mac OS X provides the following permissions:

- *Read/write/execute/append* either data or extended attributes.
- *Read/write*(regular) attributes.
- *Read/write security settings or change ownership.*
- *Add/delete files, directories, or children.*
- *Search/list directories.*

Tiger provides role-based administration, that is, a user can logon into the Administrators role and accomplish one privileged operation at a time. Various security features which are applied to all accounts:

- *Disable automatic logon*
- *Require password to unlock each secure system preference*
- *Logout after X minutes of inactivity*
- *Use secure virtual memory* means that virtual memory will be encrypted on the dics.

Chapter 6

Novel

6.1 Netware

Novel Netware is one of the oldest network operating systems. Version 3.x used bindery mode meaning that each server had a database of users. If a user wants to access a resource on a server, he needs to have an account on the server. Version 4.0 introduced Novel Directory Services (NDS) which replaced bindery mode. However, for compatibility reason bindery emulation is still possible. With NDS a user has a single login to access the resources on all the servers. Netware 5.0 brought GUI, native support for TCP/IP and virtual memory support.

Netware defines a set of access rights that are used to control access on directories and files. They are the following:

- (Read). Allows opening and reading from files or executing programs.
- (Write). Allows opening and modification of files, however does not allow (by itself) to create files.
- (Create). Allows to create files and directories. After the file was created, the user can write into it. Once the file is closed, the user can't write to it unless he has the write right. If set on a file, it can be salvaged after it has been deleted.
- (Erase). Allows to delete files and subdirectories.
- (Modify). Allows to change the attributes of files and directories, however does not allow to write to them.
- (File Scan). Grants the right to see the contents of a folder, or file (not the contents of a file).
- (Access Control). Grants the right to change the trustee assignments and the Inherited Rights Filter of the folder or file.
- (Supervisor). Grants all rights to the folder or file. Users with this right can grant or deny other users rights to the folder or file.

Rights set on a directory can inherited by its subdirectories and files that it contains. The Inherited Rights and Filters allows to filter rights, in other

words restrict the set of rights for users. That way the effective rights of a user are determined by not only the rights of a directory or file.

NetWare defines a set of directory and file attributes, some of which can only be set on files. There attributes that can be set both on files and directories are the following:

- *(H) Hidden*. Files and directories are not being shown by standard commands.
- *(Dc) Don't Compress*. If set the files are not compressed.
- *(Ic) Immediate Compress*. After the file is closed, it is immediately compressed. In a directory, as soon as every files is closed, all of them are compressed.
- *(Di) Delete Inhibit*. Files and directories cannot be deleted even if the erase right is granted.
- *(*(Dm) Don't Migrate). Files and directories are not being migrated to another storage medium.
- *(N) Normal*. It is a default attribute, which indicates the Read/Write attribute is assigned and the Shareable attribute is not.
- *(P) Purge*. As soon as a file or directory is deleted it cannot be recovered when the purge attribute is set.
- *(Ri) Rename Inhibit*. If set, the file or directory cannot be renamed.
- *(Sy) System*. Files and directories are not listed using standard commands. Marks files and directories used by the operating system.

Attributes that only apply to files are:

- *(A) Archive*. If set, the file is to be archived.
- *(Ci) Copy Inhibit*. Prevents copying the file for Macintosh users.
- *(Ds) Don't Suballocate*. If set, the data won't be suballocated.
- *(I) Index*. Files with more than 64 FAT entries will be indexed.
- *(Ro) Read Only*. The file cannot be modified. Automatically sets Delete Inhibit and Rename Inhibit on the file.
- *(Rw) Read/Write*. File is available for writing.
- *(Sh) Shareable*. The file can be accessed by more users at once.
- *(T) Transactional*. Indicates that a file can be tracked and protected by the Transaction Tracking System.
- *(X) Execute Only*. The file can be executed and renamed. However it cannot be copied or modified. The only way to clear the flag is to delete the file.

6.1.1 Accounting

Accounting under a NetWare operating system includes the following things:

- *Charge customers who use file server resources*.
- *Monitor file server usage*.

Netware provide the following accounting options:

- *Charging users for resources consumed*
- *Tracking and charging users for specific services*
- *Charging users for file server disk space*
- *Charging users for file server time*
- *Charging users for file server requests (such as read or write requests)*

A file server can charge users based on the following types of services:

- *Blocks Read* charges the user for the amount of data being read from the server. The amount to be charged for each block read can be specified. The block vary in sizes - 4KB, 8KB, 16KB, 32KB, 64KB.
- *Blocks Written* charges the user for the amount of data being written on the server.
- *Connect Time* charges the user for the amount of time being logged in to a server. The charge is specified per minute.
- *Disk Storage* charges the user for every block stored on the server. The charge is specified by block per day.
- *Service Requests* charges the user for each service requested, such as directory listing. The charge is specified per request.

Chapter 7

File Systems

A file system has to provide fast access to data, while efficiently using space. Unfortunately there is a tradeoff between these, as faster recovery of data brings more wasted space. We will however strive to discover what data protection feature filesystems offer.

7.1 FAT12, FAT16, FAT32

FAT standing for File Allocation Table is a very well known file system. Nowadays FAT is being replaced by NTFS, but with its features it suited the demand of single-user operating systems such as DOS. FAT was not designed as a file system for multiprogramming and networked operating systems. FAT contains two FAT tables which contain information about the clusters. The second file allocation table is just a copy of the first one (for security purposes), although placing it right after the first one is not really logical. If the first one gets damaged, chances are the second copy will get damaged too. FAT32 stores the copy somewhere in the middle of the partition. FAT has a hierarchical directory structure. On the top of this structure is the root directory which has a fixed position (not true on FAT32). It is located after the two FAT copies. All other directories create a treelike structure. Directories contain directory entries that are 32bytes long and contain all attributes and a pointer of a file. The pointer marks the first cluster occupied by the file. VFAT is a version of FAT that adds support for long file names, but the implementation of the long file names is somewhat weird.

7.1.1 File attributes

File attributes are stored in a single byte and provide some basic access control. They are the following:

- Read-Only - the files marked as read-only shouldn't be edited or deleted.
- Hidden - files marked as hidden should not be visible to a user using standard commands and tools.
- System - files marked as system shouldn't be edited or deleted at all as they provide the functionality of the system.
- Volume Label - it this flag is set, than the name specifies the label of the current volume

- Directory - indicates whether the file should be treated as a directory
- Archive - files marked as archive should be activated. After archiving, the files archive attribute is set to 0. If the contents of the file change, the archive bit is set to 1.

7.1.2 Timestamps

Timestamps of individual files are stored also in the directory entries. They are the following:

- time and date of the file creation
- date of the last file access
- date and time of the file's last modification

The main drawback from our point of view is that FAT does not implement any kind of access rights. This would be almost unacceptable in the multiuser and/or network operating systems. It has also a 2GB file size limit that is clearly insufficient. FAT it is mainly used with low capacity media and for data sharing for its good support among operating systems.

7.2 NTFS 4.0

NTFS standing for New Technology File System was developed for Windows NT operating system. Huge emphasis is placed on security and reliability. NTFS also supports long file names, has a more efficient storage mechanism than FAT and provides capability of managing much larger partitions and files. NTFS uses a database like approach when handling data. All structures are handled as files and all (properties and data of these) information about these files are handled as attributes.

The key system file is the Master File Table (MFT), which contains a record for every file on an NTFS partition. The first 16 records are reserved for special use. The most important for us are the following records of the MFT:

- *Master File Table (MFT)* record contains information about the MFT itself.
- *Volume Descriptor* contains information about the partition, such as NTFS version, creation time, etc.
- *Attribute Definition Table* describes types of attributes.
- *Root Directory / Folder* is a pointer to the root directory.

Every record in the MFT (representing a file) have attributes. Each attribute may be resident or nonresident. The size of each record is 1024

bytes. However some attributes are always resident, as they contain data that needs fast access or point to nonresident attributes. If the attributes of the file don't fit into the record, either a new record is allocated or the attribute is placed in an extent and becomes nonresident. A nonresident attribute is stored somewhere on the partition and the record has a pointer to it. As every information about a file is treated as an attribute, if the file is small enough it fits entirely into the record in the MFT. Each MFT record for every file and directory contains a security descriptor attribute. The security descriptor can contain two kinds of lists:

- *Discretionary Access Control List (DACL)* stores permissions for groups of users or individual users. The permissions can be set by the owner of the file.
- *System Access Control List (SACL)* is set by the administrator and determines what events occurring on the file will be audited.

The access control lists contain access control entries (ACE). ACEs contain the ID of the group or ID of the users and the permissions that will apply to them. The auditing possibilities and the set of possible permission is listed in the Windows NT 4 section.

7.3 NTFS 5.0

This version of NTFS was introduced together with Windows 2000 operating system. In this section we will only discuss the new features of this version. NTFS 5.0 adds support for disk quotas, file encryption, reparse points and an improved permission scheme. *Disk quotas* are used to keep track of how much space individual users consume. Than we can use Windows 2000's quota management to limit the space consumption. We have a few different options due to quota management:

- We can set the "warning" level and after a user consumed more space than the level specifies a warning will be generated. The "limit" level on the other hand provides a hard limit. A process trying to allocate space beyond this level will get an error message, that is, his request will be rejected.
- Quota management may be applied on a per-user or per-volume basis depending on our particular needs.
- Finally Windows 2000 provides means to monitor and log events that led to space consumption above the already mentioned levels (if set).

Windows 2000 provides transparent *file encryption*. This is particularly useful if we want to assure that even after a disc with important data had been stolen, nobody can get to the data on it. *Reparse points* are segments of code that can be inserted into files. After opening the file the operating

system recognizes the reparse point and determines what kind of action it trigers. A few thing that can be implemented by reparse points are:

- *Symbolic links* that redirect access from one file to the other.
- *Junction points* that redirect access from one directory to the other.
- *Remote Storage Servers*. Some files are automatically moved to another medium. They however leave a reparse point behind, that ensures the file will be moved back. Using reparse points the whole process of moving files from one medium to another becomes transparent.

NTFS 5.0 provides an extended set of possible permission compared to version 4. The list of permission can be found in the Windows 2000 section.

7.4 ext2

The most popular Linux file system and successor of ext filesystem. However ext2 took a lot of ideas from Berkeley's Fast File System. It supports up to 255 characters long filenames, fast symbolic links (symbolic links are stored in the inode itself). In an ext2 filesystem we can specify through a mount option whether to have BSD or SVR4 file semantics. With BSD semantics, the files are created with the same group ID as the parent directory. With System V semantic, if a directory has the set group ID bit set, new files inherit the group ID bit of the parent directories. Subdirectories also inherit the group ID and set group ID bit. Otherwise files and directories inherit the primary group ID of the calling process.

7.4.1 Inod Fields

The inode on an ext2 filesystem contains (among others) these fields:

- 16bit for file mode
- 16bit for owner uid
- 32bit for the file size
- 32bit for the last access time
- 32bit for the creation time
- 32bit for the last modification time
- 32bit deletion time
- 16bit for the group id
- 32bit for the file flags (described in the File Attributes subsection)
- 32bit for the file ACL

- 32bit for the directory ACL

On an ext2 file system, there are some reserved inodes. The ones we are interested in are:

- The 3rd and 4th are ACL inodes.
- The 6th inode is used for file undelete.

7.4.2 File Attributes

The supported attributes are:

- *EXT2_SECRM_FL* With this attribute set, whenever a file is truncated the data blocks are first overwritten with random data. This ensures that once a file is deleted, it is not possible for the file data to resurface at a later stage in another file.
- *EXT2_UNRM_FL* allows the file to be undeleted.
- *EXT2_SYNC_FL* when set, file metadata, including indirect blocks, is always written synchronously to disk after an update. This does not apply to regular file data.
- *EXT2_COMPR_FL* Marks that the file is compressed. All subsequent access must use compression and decompression.
- *EXT2_APPEND_FL* With this attribute set, a file can only be opened in append mode (O_APPEND) for writing. The file cannot be deleted by anyone.
- *EXT2_IMMUTABLE_FL* If this attribute is set, the file can only be read and cannot deleted by anyone.

7.5 ext3

A journaling file system based on ext2.

7.6 UFS

Unix File System was developed for Unix and is thus one of the most common filesystems used in Unix operating systems. UFS is based on Berkeley Fast File System (FFS). Every logical entity on a UFS is stored as a file. Each file is of a certain type, which determines how it is going to be treated (interpreted) by the OS. Each file on the UFS filesystem is represented by an index node or *inode*. The inode contains information about a particular file and pointers to its data. Some of the inode's field are the folowing:

- The type and access mode for the file. Stored on 16 bits.

- The file's owner. Stored on 16 bits.
- The group-access identifier. Stored on 16 bits.
- The time that the file was created. Stored on 32 bits.
- The time of last modification. Stored on 32 bits.
- The time when the file's inode was most recently updated by the system. Stored on 32 bits.
- The size of the file in bytes. Stored on 32 bits.
- The reference or link count. Stored on 16 bits.
- The kernel and user settable flags that describe characteristics of the file
- The number of directory entries that reference the file
- The kernel and user settable flags that describe characteristics of the file
- The size of the extended attribute information

The filenames are stored in directories rather than in inodes and can be 255 files in length. One inode can have multiple names and each time a new filename is associated with the inode, the inode's reference counter is incremented. When a name is deleted, the counter in the inode is decremented. When the counter's value reaches zero the inode is deleted from the medium.

The type and access mode field of an inode occupies 16 bits and has the following format:

- The 4 most significant bits indicate the file type
  - 0x1 Fifo 'p'
  - 0x2 Character special files 'c'
  - 0x4 Directory 'd'
  - 0x6 Block special files 'b'
  - 0x8 Regular file '-'
  - 0xA Symbolic link 's'
  - 0xC Socket
  - 0xD Door
- Access rights are specified for user, group and others as read, write and execute. This uses 9 bits.
- The most significant 3 bits of the mode information specify

- Set user and set group id
- enable mandatory record/file locking (for non-executable files)
- stickiness (save text image)
- allow deletion of non-owned files in directory

File access rights determine whether the following operations can be performed on files. The ”Read” permission allows to read the contents of a file. The ”Write” permission allows to write into the file and delete it. The ”Execute” permissions allow to execute the file.

Directory access rights have different meanings then access rights for regular files. ”Read” right means that the directory can be read, that is list it's contents. ”Write” right means you can create and delete files and directories in the directory. ”Execute” right means that the user cannot enter the directory, list files and read contents of the files. ”Sticky” right means that you can create files and directories, you can delete only files that you own. Used for the ”tmp” directory.

File flags define characteristics of files, such as append-only, immutable and not needing to be dumped. An *immutable* file cannot be changed, moved or deleted. An *append-only* file cannot be moved or deleted, and data can be appended to it. These properties can be set by the owner of the file or by the superuser and are positioned on the low 16 bits. The higher 16 bits contain flags to mark the file as append-only and immutable. However, only the superuser can set these properties and cannot be cleared when the system is secure.

7.7 UFS2

UFS2 is the successor of UFS filesystem. It addresses some of the ”limitations” of UFS. The size of a UFS inode is 128 bytes, in UFS2 the size has been increased to 256 bytes. The time fields have been expanded from 32 bits to 64 bits. The 32 bits time fields would in the year of 2038 overflow, so this is a very practical innovation. A new time field has been added that is set when the inode is allocated and is not changed afterward. The file flags field has been expanded to 64 bits, the lower 32 bits remain the same as for UFS1, the upper 32 bits are managed by the kernel. They contain kernel flags such as SNAPSHOT flag and OPAQUE flag. UFS preallocated all of its inodes during the filesystem creation and this could take a lot of time. UFS2 uses *dynamic inodes* which means that only a set of inodes is preallocated. When the system runs out of free inodes it dynamically creates a set of new ones (typically 32 or 64).

However the most useful feature from our point of view is support for *extended attributes*. Extended attributes can be used to store data totally separated from the contents of the file. Actually there are five 64-bit pointers free, but only two are available for use as extended attributes. The addi-

tional three are left as reserved for future purposes. The typical use of extended attributes is the implementation of access control lists.

7.8 HFS+

Hierarchical File System Plus was designed for Mac OS X. It supports journaling, quotas, byte-range locking, etc. Mac OS X is very similar to BSD systems and the security model on a filesystem level reflects it. So basically we have the we can set ”Read”, ”Write” and ”Execute” permissions for the ”Owner”, ”Group” and the ”Others”. Just like in a standard Unix-like operating system. HFS+ however supports access control lists. HFS+ stores modification and creation dates with files.

7.9 HPFS

High Performance File System is used with OS/2. Each file and directory is represented by a fnode. Filenames can be up to 254 characters long. HPFS implements the same file and directory attributes as FAT, but adds a feature called *extended attributes*. If the extended attributes associated with files or directories are small enough, they are stored in the fnode. If they do not fit in the fnode, the extended attributes are stored in a separate run.

7.10 XFS

XFS was developed by SGI for their IRIX operating systems. It uses allocation groups which basically divide the filesystem into parts. These parts contain every information needed to store data in the allocation group and no access has to be made anywhere else. This makes parallelism possible as interactions with more allocation groups can be made simultaneously. XFS also supports journaling to address data loss issues. XFS supports user and group quotas. XFS considers quota information as filesystem metadata. [9] XFS also supports POSIX access control lists.

7.11 JFS

JFS standing for Journaled File System is a journaling file system created by IBM and is used in AIX operating systems. Each partition contains one aggregate. The aggregate stores all the necessary information such as primary and secondary aggregate superblock, aggregate inode table, aggregate inode map, etc. A fileset is a set of files and directories that form an independently mountable sub-tree. [8] There can be more than one fileset in an aggregate. Basically JFS uses inodes to store files and directories. The number of inodes is fixed and determined when the filesystem is created. Every inode can have an access control list associated with it. Access control list can represent permissions, user identifiers or group identifiers. An ACL

must fit within 8192 bytes. Directories may have two access control list associated with them: the initial directory ACL which applies to the directory itself and the initial file ACL which is applied to files created within the directory. JFS does not specify how to store ACLs. However there is a file present in each fileset that contains the ACLs. This file is represented with inode number one. Each node that has an ACL associated with it stores an index number into the ACL file. The ACL consists of a bitmap and the actual access control list entries. The bitmap is used to indicate which ACEs are free for use. The ACL file contains a 4K bitmap followed by 8M ACL entries. One bit in the bitmap represents 256 bytes of contiguous disk space. JFS also provides *extended attributes* that allow to specify and attach additional data to JFS objects. Extended attributes can be stored either directly in the inode or separately. There can be only one extent of extended attributes per one inode.

Chapter 8

Utility

The practical part of this thesis addresses some development issues around a utility for transferring permissions of files and directories among operating systems. The supported operating systems are some versions of Unix, Linux and Windows operating systems. When transferring permissions across various operating systems, we have to face up a few very difficult problems. What is worse, we cannot come up with an ideal solution, because the differences between operating systems are substantial. There are a few things we would like to point out sections where some necessary compromises are to be made:

- *The user/group enviroment cannot be transferred in a way, that would assure the possibility for everyone to have the same options concerning access control.*
- *Even the permission numbers and types may differ, which absolutely eliminates to adapt the same access control features.*
- *The semantics are another thing that we cannot change (only partially) in operating systems. Different procedures may have different results although not always noticeable.*

These are just some points that limit us. Perhaps the most noticable effect is the "disappearance" of permissions, when for instance transferring files from Windows to Unix or Linux. We strive to provide the most intuitive, effective, yet most simple way to store and set permissions. We rather provide the a utility with very basic functionality, but demonstrates the possibilities and also the difficulties of such thing as transferring permissions. Naturally under all operating systems we need to be logged in as Administrators (Superusers). Some (or even all) problems I mentioned may not affect a particular permission transfer. However it is obvious that under real conditions, while transferring a huge amount of files and directories which are owned by hundreds of users, the mentioned problems will arise, even if unnoticed for some time. The utility allows us to store the permissions of a directory structure, which can also be the root directory. All permissions are stored in one file in easy to read textual representation. The way the permissions are stored is way from effective. However, this allows us to see the files as a collection of attributes - the path, filename, owner, group and permissions. Even if part of the file gets damaged, we can still restore every setting for

files that were not striken by the damage. We can also manually change the attributes that are stored for individual files.

8.1 Unix and Linux

Unix and Linux provide very basic access control for files and directories compared to Windows 2k (and upper) operating systems. This makes setting permissions more less straightforward. Even though some version of Unix and Linux do support ACLs which would allow to more precisely set permissions, we decided to use only the basic "ugo" model. The reason is portability, as not all Unix (Linux) versions support them. Naturally we store the sticky bit, but we do not store the setuid and setgid flags.

8.2 Windows 2k

We focused only on version Windows 2000 and newer, because they provide very extensible features concerning access control. Also we tried to make thing work as simply as possible. It was more difficult to decide how to filter the permissions to be able to apply them on Unix-like operating systems. This filtering occurs not only as the loss of granularity of access control, but sometimes the most of the permissions are simply thrown away. For the most part Unix-like access control is a subset of that of Windows operating systems. However, we must add that this is only true from our point of view, when programming the utility.

Chapter 9

Conclusion

We have taken a look at the access control in a variety of different operating systems. Sometimes we may have taken a very straightforward description of ways to control access. But we believe that seeing all the actual possibilities allows us to understand the evolution of security in computer environments better. And we can notice the huge leaps in security from operating system to operating system. If we think into it, we already have smart ways and tools that aid us when building secure networks (or environments generally). Even if an administrator has to administer permissions for hundreds of users in huge directory trees, we are getting nearer to a point. To a point where we already have the means to solve things in a short amount of time, but we haven't learned to use them, because the technology is so rapidly changing. We have to remark that the thesis covers access control mechanism rather on the surface. It would be possible to describe these mechanism in a very different manner. On the level of system calls, operating system component operations, kernel structures, etc. Moreover, the most up to date operating systems provide such extensible possibilities, that even a whole thesis wouldn't exhaust them. Much more could be said about access control, but the object of the thesis was to provide an overview of a variety of different operating systems. As some of these operating systems were relatively old and the there wasn't such an emphasize on access control, some of the text may seem out of date. But we really think that by knowing where we came from, can we really tell where we are heading.
The practical part of thesis shows that the differences between various operating systems grow. The enviroment created in operating systems seems to be so unique, that we don't have the chance to recreate that enviroment in a different operating system. It is no more just a matter of files being being placed on the filesystem and the ability to run certain application. Utilising networks on the other hand, helps us through built in utilities to make use of different operating system services transparently. The amount of freedom, resources and services provided by network comes however at a price. The price being the necessity to protect our data. Access control lists are starting to be extensively used in Unix-like operating systems and that way the gaps are likely to close a little. However additional features are incorporated and comparing operating systems becomes more and more dificult. They all provide a stable, secure enviroment, but are all diferent.

Bibliography

[1] Steve D. Pate *UNIX Filesystems: Evolution, Design, and Implementation(VERITAS Series)*. Indianapolis: Wiley Publishing, 2003.

[2] *Microsoft Security Bulletin (MS00-072)*. Document available on URL http://www.microsoft.com/technet/security/Bulletin/MS00-072.mspx (may 2006)

[3] *Windows95/98 Resource Kit*.
Document available on URL www.winresource.com (may 2006)

[4] *User Manager for Domains Help*. Document available in Windows NT

[5] David A. Solomon, Mark E. Russinovich *Inside Microsoft Windows 2000 (Third edition)*. Redmond, Washington 98052-6399: Microsoft Press, 2000

[6] *Novell Documentation*, 2005
Document available on URL http://www.novell.com/documentation//nw51/index.html?page=/documentation//nw51/trad_enu/data/h8gdk9xq.html (may 2006)

[7] *MSDN Library*, 2006
Document available on URL
http://msdn.microsoft.com/library/default.asp?url=/library/en-us/secauthz/security/ace_header.asp (may 2006)

[8] S. Best & D. Kleikamp, *How the Journaled File System handles the on-disk layout*. 2003. Document available on URL http://www-128.ibm.com/developerworks/linux/library/l-jfslayout (may 2006)

[9] *XFS: A high-performance journaling filesystem*.
Document available on URL http://oss.sgi.com/projects/xfs (may 2006)

[10] *Frank Bodammer and others: SUSE LINUX - Administration Guide*.
Document available on URL http://www.novell.com/documentation/suse91/suselinux-adminguide/html/index.html (may 2006)

[11] Daniel P. Bovet, Marco Cesati: *Understanding the Linux Kernel, Second Edition*. USA, O'Reilly & Associates, 2003

[12] Kirk McKusick, George Neville-Neil: *The Design and Implementation of the FreeBSD Operating System*. Addison Wesley, 2004