SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **00**:1–7 *Prepared using* speauth.cls *[Version: 2000/03/16 v2.12]*

Access control: principles and solutions

Sabrina De Capitani di Vimercati[1], Stefano Paraboschi[2], Pierangela Samarati[3,*]

[1] *Università di Brescia – Dipartimento di Elettronica per l'Automazione*
[2] *Politecnico di Milano – Dipartimento di Elettronica e Informazione*
[3] *Università di Milano – Dipartimento di Tecnologie dell'Informazione*

SUMMARY

Access control is the process of mediating every request to resources and data maintained by a system and determining whether the request should be granted or denied. The variety and complexity of the protection requirements that may need to be imposed makes access control a far from trivial process. Expressiveness and flexibility are top requisites for an access control system together with, and usually in conflict with, simplicity and efficiency. In this paper, we discuss the main desiderata for access control systems and illustrate the main characteristics of access control solutions in some of the most popular existing systems.

1. Introduction

An important requirement of any system is to protect its *data* and *resources* against unauthorized disclosure (*secrecy* or *confidentiality*) and unauthorized or improper modifications (*integrity*), while at the same time ensuring their availability to legitimate users (*no denial-of-service* or *availability*) [10]. The problem of ensuring protection has existed since information has been managed. However, as technology advances and information management systems become more and more powerful, the problem of enforcing information security also becomes more critical. The increasing development of information technology in the past few years, which has led to the widespread use of computer systems to store and manipulate information and greatly increased the availability and the processing and storage power of information systems, has also posed new serious security threats and increased the potential damage that violations may cause. Organizations more than ever today depend on the information they manage. A violation to the security of the information may jeopardize the whole system working and cause serious damages. Hospitals, banks, public administrations,

*Correspondence to: Pierangela Samarati, Dipartimento di Tecnologie dell'Informazione – Università di Milano. Via Bramante, 65 – 26013 - Crema, Italy. Email: samarati@dti.unimi.it.

Received received
Revised revised

private organizations, all of them depend on the accuracy, availability, and confidentiality of the information they manage. Just imagine what could happen, for example, if an organization's data were improperly modified, weren't available to the legitimate users because of a violation blocking access to the resources, or were disclosed to the public domain.

A fundamental component in enforcing protection is represented by the *access control* service whose task is *to control every access to a system and its resources and ensure that all and only authorized accesses can take place*.

To this purpose, every management system usually includes an access control service that establishes the kinds of regulations (policies) that can be stated, through an appropriate specification language, and then enforced by the access control mechanism enforcing the service. By using the provided interface, security administrators can specify the access control policy (or policies) that should be obeyed in controlling access to the managed resources.

The definition of access control policies to be fed into the access control system is far from being a trivial process. One of the major difficulties lies in the interpretation of, often complex and sometimes ambiguous, real world security policies and in their translation in well defined unambiguous rules enforceable by the computer system. Many real world situations have complex policies, where access decisions depend on the application of different rules coming, for example, from laws practices, and organizational regulations. A security policy must capture all the different regulations to be enforced and, in addition, must consider all possible additional threats due to the use of computer systems. Given the complexity of the scenario, it is therefore important that the access control service provided by the computer system be expressive and flexible enough to accommodate all the different requirements that may need to be expressed, while at the same time be simple both in terms of use (so that specifications can be kept under control) and implementation (so to allow for its verification).

An access control system should include support for the following concepts/features:

- *Accountability and reliable input*. Access control must rely on a proper input. This simple principle is not always obeyed by systems allowing access control rules to evaluate on the basis of possibly unreliable information. This is, for example, the case of *location-based* access control restrictions, where the access decision may depend on the IP from which a request originates, an information which can be easily faked in a local network, thus fooling access control (allowing non legitimate users to acquire access despite the proper rule enforcement). This observation has been traditionally at the basis of requiring proper user authentication as a prerequisite for access control enforcement [13]. While more recent approaches may remove the assumptions that every user is authenticated (e.g., by allowing credential-based access control), still the assumption that the information on which access decision is taken must be correct indeed continues to hold.
- *Support for fine- and coarse-specifications*. The access control system should allow rules to be referred to specific accesses, providing fine-grained reference to the subjects and objects in the system. However, fine-grained specifications should be supported, but not forced. In fact, requiring the specification of access rules with reference to every single user and object in the system would make the administration task a heavy burden. Beside, groups of users and collections of objects often share the same access control requirements. The access control system should then provide support for authorizations

specified for groups of users, groups of objects, and possibly even groups of actions [9]. Also, in many organizational scenarios, access needs may be naturally associated with *organizational activities*; the access control system should then support authorizations referred to organizational roles [14].

- *Conditional authorizations*. Protection requirements may need to depend on the evaluation of some conditions [10]. Conditions can be in the simple form of system's predicates, such as the date or the location of an access (e.g., 'Employee can access the system *from 9 am to 5 pm*'). Conditions can also make access dependent on the information being accessed (e.g., 'Managers can read payroll data of *the employees they manage*').
- *Least privilege*. The least privilege principle mandates that every subject (active entity operating in the system) should always operate with the least possible set of privileges needed to perform its task. Obedience of the least privilege requires both static (policy specification) and dynamic (policy enforcement) support from the access control system. At a static level, least privilege requires support of *fine-grained authorizations*, allowing granting each specific subject only those specific accesses it needs. At a dynamic level, least privilege requires restricting processes to operate within a *confined set of privileges*. Least privilege is partially supported within the context of *roles*, which are essentially privileged hats that users can take and leave [14]. Authorizations granted to a role apply only when the role is active for a user (i.e., when needed to perform the tasks associated with the role). Hence, users authorized for powerful roles do not need to exercise them until those privileges are actually needed. This minimizes the danger of damages due to inadvertent errors, or by intruders masquerading as legitimate users. Least privilege also requires the access control system to discriminate between different processes, even if executed by the same user, for example, by supporting authorizations referred to programs or *applicable only within the execution of specific programs*.
- *Separation of duty*. Separation of duty refers to the principle that no user should be given enough privilege to misuse the system on their own [11]. While separation of duty is better classified as a policy specification constraint (i.e., a guideline to be followed by those in charge of specifying access control rules), support of separation of duty requires the security system to be expressive and flexible enough to enforce the constraints. At a minimum, fine-grained specifications and least privilege should be supported; *history-based authorizations*, making one's ability to access a system dependent on previously executed access, are also a convenient means to support separation of duty.
- *Multiple policies and exceptions*. Traditionally, discretionary policies have been seen as distinguished into two classes: *closed* and *open* [10]. In the most popular closed policy, only accesses to be authorized are specified; each request is controlled against the authorizations and allowed only if an authorization exists for it. By contrast, in the open policy, (negative) authorizations specify the accesses that should not be allowed. All access requests for which no negative authorization is specified are allowed by default.
- *Policy combination and conflict-resolution*. If multiple modules (e.g., for different authorities or different domains) exist for the specification of access control rules, the access control system should provide a means for users to specify how the different modules should interact, for example, if their union (maximum privilege) or their

intersection (minimum privilege) should be considered. Also, when both permissions and denials can be specified, the problem naturally arises of how to deal with *incompleteness*, that is, existence of accesses for which no rule is specified, and *inconsistency*, that is, the existence of accesses for which both a denial and a permission are specified. Dealing with incompleteness–requiring the authorizations to be complete would be very impractical–requires support of a *default* policy either imposed by the system or specified by the users. Dealing with inconsistencies require support for *conflict resolution* policies. Different conflict resolution approaches can be taken, such as the simple *denials take precedence* (in the case of doubt access is denied), or *most specific* criteria that make the authorization referred to the more specific element (e.g., a user is more specific than a group, and a file is more specific than a directory) take precedence. While, among the different conflict resolution policies that can be thought of (see [10] for a deeper treatment), some solutions may appear more natural than others, none of them represents "the perfect solution". Whichever approach we take, we will always find one situation for which the approach does not fit. Therefore any conflict resolution policy imposed by the access control mechanism itself will always result limiting. On the other side, support of negative authorizations does not come for free, and there is a price to pay in terms of authorization management and less clarity of the specifications. However, the complications brought by negative authorizations are not due to negative authorizations themselves, but to the different semantics that the presence of permissions and denials can have, that is, to the complexity of the different real world scenarios and requirements that may need to be captured. There is therefore a trade-off between expressiveness and simplicity. Consequently, current systems try to keep it simple by adopting negative authorizations for exception support, imposing specific conflict resolution policies, or supporting a limited form of conflict resolution.

- *Administrative policies*. As access control systems are based on access rules defining which accesses are (or are not) to be allowed, an administrative policy is needed to regulate the specification of such rules, that is, define who can add, delete, or modify them. Administrative policies are one of the most important, though less understood, aspects in access control. Indeed, they have usually received little consideration, and, while it is true that a simple administrative policy would suffice for many applications, it is also true that new applications (and organizational environments) would benefit from the enrichment of administrative policies. In theory, discretionary systems can support different kinds of administrative policies: *centralized*, where a privileged user or group of them is reserved the privilege of granting and revoking authorizations; *hierarchical/cooperative*, where a set of authorized users is reserved the privilege of granting and revoking authorizations; *ownership*, where each object is associated with an owner (generally the object's creator) who can grant to and revoke from others the authorizations on its objects; and *decentralized*, where, extending the previous approaches, the owner of an object (or its administrators) can delegate other users the privilege of specifying authorizations, possibly with the ability of further delegating it. For its simplicity and large applicability the ownership policy is the most popular choice in today's systems (see Section 2). Decentralized administration approaches can be instead found in the database management system contexts (see Section 3). Decentralized administration is

convenient since it allows users to delegate administrative privileges to others. Delegation, however, complicates the authorization management. In particular, it becomes more difficult for users to keep track of who can access their objects. Furthermore, revocation of authorizations becomes more complex.

In the remainder of this paper we survey the access control services provided by some of the most popular operating systems (Section 2), database management systems (Section 3), and network solutions (Section 4). While clearly their characteristics will vary from one class to the other as different is their focus (e.g., database management systems focus on the data and rely on the operating systems for low level support), it will be interesting to see how they accommodate (or do not accommodate) the features introduced above. Also, it will be noticed how, while covering a feature in some way, some systems take unclean solutions which may have side effects in terms of security or applicability, aspects which then should be taken into account when using the systems.

2. Access control in operating systems

We describe access control services in two of the most popular operating systems: Linux (e.g., `www.redhat.com`, `www.linux-mandrake.com`, `www.suse.com`) and Microsoft Windows 2000/XP (`www.microsoft.com`).

2.1. Access control in Linux

We use Linux as a modern representative of the large family of operating systems deriving from Unix. We signal the features of Linux that are absent in other operating systems of the same family.

Apart from specific privileges like access to protected TCP ports, the most significant access control services in Linux are the ones offered by the file system. The file system has a central role in all the operating systems of the Unix family, as files are used as an abstraction for most of the system resources.

2.1.1. User identifiers and group identifiers

Access control is based on a user identifier (UID) and group identifier (GID) associated with each process. A UID is an integer value unique for each username (login name), where the association between usernames and UIDs is described in `/etc/passwd`. A user connecting to a Linux system is typically authenticated by the `login` process, invoked by the program managing the communication line used to connect to the system (`getty` for serial lines, `telnetd` for remote telnet sessions). The `login` process asks the user for a username and a password and checks the password with its hash stored in read-protected file `/etc/shadow`; a less secure and older alternative stores hashed password in the readable-by-all file `/etc/passwd`. When authentication is successful, the `login` process sets the UID to that of the authenticated user, before starting an instance of the program described in the user entry in `/etc/passwd`

(typically a shell, like `/bin/bash`). Users in Linux are members of groups. Every time a user connects to the system, together with the user identifier (UID), a primary group identifier (GID) is set. The primary GID value to use at login is defined in file `/etc/passwd`. Group names and additional memberships in groups are defined in file `/etc/group`. Command *newgrp* allows users to switch to a new primary GID. If a user is listed in `/etc/group` as belonging to the new group, the request is immediately executed; otherwise, for groups having a hashed password in `/etc/group`, the primary GID can be changed after the password has been correctly returned. However, group passwords are deprecated, as they easily lead to lapses in password management.

Processes are usually assigned the UIDs and GIDs of the parent processes, which implies that processes acquire the UIDs and GIDs associated with the login names with which sessions have been started. The process UID is the central piece of information for the evaluation of the permitted actions. There are many operations in the system that are only allowed to a user which has zero as the value of its UID. By convention, root is associated with value zero, representing the owner of the system who supervises all the activities. For instance, the TCP implementation allows only user *root* to open ports below 1024.

2.1.2. Files and privileges

In the Linux file system, each file is associated with a UID and a GID, which typically represent the UID and primary GID of the process that created the file. UID and GID associated with a file can be changed by commands *chown* and *chgrp*. Each file has an associated a list of nine privileges: *read*, *write* and *execute*, each defined three times, at the level of *user*, *group*, and *other*. The privileges defined at the level of *user* are the actions that will be permitted to processes having the same UID as the file; the privileges defined at the level of *group* are granted to processes having the same GID as the file; the privileges at the level of *other* are for processes that share neither UID nor GID with the file. The privileges are commonly represented by the *ls -l* command as a string of nine characters (preceded by a character representing the file type). Each privilege is characterized by a single letter: `r` for *read*; `w` for *write*; `x` for *execute*; the absence of the privilege is represented by character `-`. The string presents first the *user* privileges, then *group*, and finally *other*. For instance, `rwxr-xr--` means that a process having as UID the same UID as the file has all the three privileges, a process with the same GID can read and execute (but not write) the file, and remaining processes can only read the file. When a user belongs to many groups, all the corresponding GIDs are stored in a list in the process descriptor beside the primary GID; other operating systems of the Unix family stored only the primary GID within the process descriptor. All the groups in the list are then considered: if the GID of the file is the primary GID, or appears anywhere in the list, group access privileges apply.

The semantics of privileges may be different depending on the type of the file on which they are specified. In particular, for directories, the *execute* privilege represents the privilege to access the directory; *read* and *write* privileges permit respectively to read the directory content and to modify it (adding, removing and renaming files). Privileges associated with a file can be updated via the *chmod* command, which is usable by the file owner and by user *root*.

2.1.3. Additional security specifications

The file system offers three other privileges on files: *save text image* (*sticky bit*), *set user ID* (*setuid*), and *set group ID* (*setgid*). The *sticky bit* privilege is useful only for directories, where it allows only the owner of the file, owner of the directory, and root to remove or rename the files contained in the directory, even if the directory is writable by all. The *setuid* and *setgid* privileges are particularly useful for executable files, where they permit to set the UID or GID of the process that executes the file to that of the file itself. These privileges are often used for applications that require a higher level of privileges to accomplish their task (e.g., users change their passwords with the *passwd* program, which needs read and write access on file `/etc/shadow`). Without the use of these bits, enabling a process started by a normal user to change the user's password would require explicitly granting the user the write privilege on the file `/etc/shadow`. Such a privilege could, however, be misused by users who could access the file though different programs and tamper with it. The *setuid* and *setgid* bits, by allowing the *passwd* program to run with root privilege, avoid such security exposure. It is worth noticing that, while providing a necessary security feature, the *setuid* and *setgid* solutions are themselves vulnerable as the specified programs run with root privileges – in contrast to the least privilege principle, they are not confined to the accesses needed to execute their task – and it is therefore important that these programs be *trusted* [10].

The *ext2* and *ext3* file systems, the most common in Linux implementations, offer additional boolean attributes on files. Among them, there are attributes focused on low-level optimizations (e.g., a bit requiring a compressed representation of the file on the disk) and two privileges that extend access control services: *immutable* and *append-only*. The *immutable* bit specifies that no change is allowed on the file; only the user *root* can set or clear this attribute. The *append-only* bit specifies that the file can be extended only by additions at the end; this attribute can also be set only by *root*. Attributes are listed by command *lsattr* and are modified by command *chattr*.

2.1.4. IP-based security

Linux offers several utilities that base authentication on IP addresses. All of these solutions should be used with care, as IP addresses can be easily spoofed in a local network [3] and therefore fool the access control system. For this reason, they are not enabled by default. Among these utilities, *rsh* executes remote shells on behalf of users; *rcp* executes copies involving the file systems of machines in a network; NFS permits to share portions of the file system on the network. For these applications, access control typically is based on a few relatively simple textual files, which describe the computers that can use the service and the scope of the service; patterns can be used to identify ranges of names or addresses, and groups may be defined, but overall the access control features are basic. Secure solutions of the above applications (like the *ssh* application and the *scp* program offered within the same package) offer a greater degree of security, at the expense of computational resources, configuration effort and in general less availability.

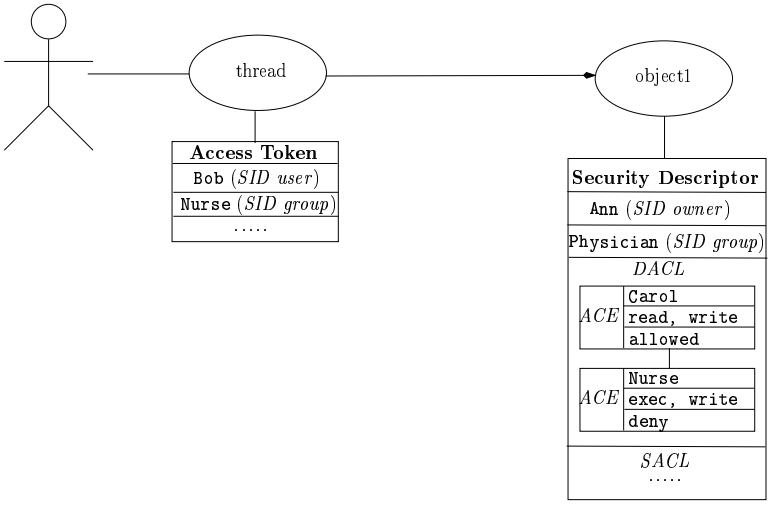

Figure 1. Access control in Windows

2.2. Access control in Windows

We now describe the characteristics of the access control model of the Microsoft Windows 2000/XP operating system (`msdn.microsoft.com`). Most of the features we present were already part of the design of Microsoft Windows NT; we will clarify the features which were not present in Windows NT and were introduced in Windows 2000. We use the term Windows to refer to this family of operating systems. We do not consider the family of Windows 95/98/ME operating systems.

2.2.1. Security descriptor

One of the most important characteristics of the Windows operating system is its object-oriented design. Every component of the system is represented as an object, with attributes and methods. In this scheme, it is natural to base access control on the notion that objects can be *securable*, that is, they can be characterized by a *security descriptor* which specifies the security requirements of the object (this corresponds to implementing access control with an access control list approach [10], equivalent to what the nine-character string is in Unix). Almost all of the system objects are *securable*: files, processes, threads, named pipes, shared memory areas, registry entries, and so on. The same access control mechanism applies to all of them.

Any subject that can operate on an object (user, group, logon session, etc.) is represented in Windows by a *Security Identifier* (SID), with a rich structure that manages the variety of

active entities. The main components of the *security descriptor* are the SIDs of the owner and of the primary group of the object, a *Discretionary Access Control List* (DACL), and a *System Access Control List* (SACL).

2.2.2. Access Control Element

Each access control list consists of a sequence of *Access Control Elements* (ACEs). An ACE is an elementary authorization on the object with which it is associated by way of the ACL; the ACE describes the subject, the action, the type (allow, deny or audit) and several flags (to specify the propagation and other ACE properties). The subject (called *trustee* in Windows) is represented by a SID. The action is specified by an *access mask*, a 32-bit vector (only part of the bits are currently used; many bits are left unspecified for future extensions). Half of the bits are associated with access rights valid for every object type; these access rights can be divided into three families:

- *generic*: *read*, *write*, *execute*, and the union of all of them;
- *standard*: *delete*, *read_control* (to read the security descriptor), *synchronize* (to wait on the object until a signal is generated on it), *write_dac* (to change the DACL), and *write_owner* (to change the object's owner);
- *SACL*: *access_system_security* (a single access right to modify the SACL; the right is not sufficient, as the subject must also have the SE_SECURITY_NAME privilege). It cannot appear in a DACL.

The remaining 16 bits are used to represent access rights specific to the object type (directory, file, process, thread, etc.). For instance, for directories access rights *open*, *create_child*, *delete_child*, *list*, *read_prop*, and *write_prop* apply. Active directory services, described in Section 2.2.8, are the base for the introduction of object-specific ACEs.

2.2.3. Access Token

Each process or thread executing in the system is associated with an *Access Token*, an object that describes the security context. An access token describing the user is created after the user has been authenticated and it is then associated with every process executing on behalf of the user. The access token contains: the SID of the user's account; SIDs of the groups which have the user as a member; a *logon* SID identifying the current logon session; a list of the privileges held by the user or the groups; an owner SID; the SID of the primary group; and the default DACL, to use when a process creates a new object without specifying a security descriptor. In addition, there are other components that are used for changing the identifiers associated with a process (called *impersonation* in Windows), and to apply restrictions.

2.2.4. Evaluation of ACLs

When a thread makes a request to access an object, its Access Token is compared with the DACL in the security descriptor. If the DACL is not present in the security descriptor, the system assumes that the object is accessible without restrictions. Otherwise, the ACEs in the

DACL are considered one after the other, and for each one the user and group SIDs in the access token are compared with the SID in the ACE. If there is a match, the ACE is applied. Order in the DACL is extremely important. The first ACE that matches will apply or deny the access rights in it. The following matching ACEs will only be able to allow or deny the remaining access rights. If the analysis of the DACL terminates and no allow/deny has been obtained for a given access right, the system assumes that the right is denied (closed policy). As an example, with reference to Figure 1, for user `Bob` the second ACE will apply (as it matches the group in the thread's access token) denying `Bob` the `execute` and `write` accesses on object1. The approach of applying the first ACE encountered corresponds to the use of a 'position-based' criterion for resolving possible conflicts [10]. While simple, this solution is quite limiting. First, it gives the users specifying the policy the complete burden of solving each specific conflict that may arise (not allowing them to specify generic high level rules for that). Second, it is not suitable if a decentralized administration (where several users can specify authorizations) should be accommodated. Also, users should have explicit direct write privilege on the DACL to properly order the ACEs. However, doing so it would be possible for them to abuse the privilege and set the ACL in an uncontrolled way.

It is worth noting how an empty DACL (which returns no permissions) will deny all users the access to the object, whereas a null DACL (which returns no restrictions) would grant them all. Then, attention must be paid to the difference between the two.

The DACL is set by the object creator. When no DACL is specified, the default DACL in the access token is used by the system. The SACL is a sequence of ACEs like the DACL, but it can only be modified by a user having the administrative privilege SE_SECURITY_NAME, and describes the actions that have to be logged by the system (if the ACE for a given right and SID is positive, the action must be logged; if it is negative, no trace will be kept); the access control system records access requests that have been successful, rejected, or both, depending on the value of the flags in the ACE elements in the SACL. Each monitored access request produces a new entry in the *security event log*.

2.2.5. System privileges

A system privilege in Windows is the right to execute privileged operations, such as making a backup, debugging a process, increasing the priority of a process, increasing the quota, and creating accounts. All these operations are not directly associated with a specific system object, and they cannot conveniently be represented by ACEs in an object security descriptor. System privileges can be considered as authorizations without an explicit object.

System privileges can be associated with users and groups accounts. When a user is authenticated by the system, the access token is created; the access token contains the system privileges of the user and of the groups in which the user is a member. Every time a user tries to execute a system privileged operation, the system checks if the access token contains the adequate system privilege. System privileges are evaluated locally; a user can then have different system privileges on different nodes of the network.

2.2.6. Impersonation and restricted tokens

Impersonation is a mechanism that permits threads to acquire the access rights of a different user. This feature is similar to the *setuid* and *setgid* services of Linux, where the change in user and group identifier permits programs invoked by a user to access protected resources. In Windows, impersonation is also an important tool for client-server architectures. The server uses impersonation to acquire the security context of the client when a request arrives. The advantage is that, in a network environment, a user will be able to consistently access the resources for which the user is authorized, and the system will be better protected from errors in the protocol used for service invocation or in the server application.

Each process has an access token (created at logon) built from the profile of the authenticated user. An impersonating thread has two access tokens, the primary access token that describes the access token of the parent process, and the impersonation access token that represents the security context of a different user. Obviously, impersonation requires an adequate system privilege.

Windows 2000 introduced primitives for the creation of restricted tokens. A restricted token is an access token where some privileges have been removed, or restricting SIDs have been added. A restricting SID is used to limit the capabilities of an access token. When an access request is made and the access token is compared with the ACEs in the ACL, each time there is a match between the restricting SID in the token and the SID in an ACE, the ACE is considered only if it denies access rights on the object.

2.2.7. Inheritance

Some important securable objects contain other securable objects. As an example, folders in the NTFS file system contain files and other folders; registry keys contain subkeys. This containment hierarchy puts in the same containers objects that are often characterized by the same security requirements. The hierarchy results then extremely convenient permit an automatic propagation of security descriptions from an object to all the objects contained within it (cf. Section 1, support of abstractions). This feature is realized in Windows by access control *inheritance*.

A difference exists between Windows NT and Windows 2000 and later with respect to inheritance. In Windows NT there was no distinction between direct and inherited ACEs; also ACEs were inherited by an object only when the object was created or when a new ACL was applied onto an object. The result was that a change in an ACE was not propagated down the hierarchy to the object that had inherited it. In Windows 2000 and later, propagation is automatic (as users would probably expect). Also, Windows 2000 gives higher priority to ACEs directly defined on the specific objects, by putting the inherited ACEs at the end of the DACL.

In Windows 2000 and later, every ACE is characterized by three flags. The first flag is active if the ACE has to be propagated to descendant objects. The second and third flag are active only if the first flag is active. The second flag is active when the ACE is propagated to children objects without activating the first flag, thus blocking propagation to the first level. The third flag is active when the ACE is not applied to the object itself. In addition, in the

security descriptor of a securable object there is a flag that permits to disable the application of inherited ACEs to the object.

2.2.8. Fine granularity access control

Another innovation of Windows 2000 is the introduction of a fine graines access model, which supports the Windows object model. There are two different solutions. The first solution is applicable to directory services objects and uses new ACE types defining access rights on specific object properties. These ACEs are based on an object structure that extends the regular ACE with two GUID parameters (a GUID is the general object identifier). The first GUID represents the specific property, property set, or child object for which the ACE is defined. The second represents the object that can inherit the ACE.

The second solution is the one offered within Active Directory services by the *controlAccessRight* object. The object specifies access rights on object properties or on user-defined actions. The object is then referenced within an ACE inserted in the DACL of the object itself.

3. Access control in database management systems

Database management systems (DBMSs) usually provide access control services in addition to those provided by the underlying operating systems [4]. DBMS access control allows references to the data model concepts and the consequent specification of authorizations dependent on the data and on the applications. Most of the existing DBMSs (e.g., Oracle Server, SQL Server, Postgres) are based on the relational data model and on the use of SQL (Structured Query Language) as the Data Definition and Manipulation language [2]. The SQL standard provides commands for the specification of access restrictions on the objects managed by the DBMS. We here illustrate the main SQL facilities with reference to the latest version of the language, namely SQL:1999 [6].

3.1. Security features of SQL

SQL access control is based on *user* and *role* identifiers. User identifiers correspond to login names with which users open the DBMS sessions. DBMS users are defined by the DMBS in an implementation-dependent way and are usually independent of the usernames managed by the operating system; SQL does not define how OS users are mapped to SQL users. Roles, introduced in SQL:1999, are "named collections of privileges" [12], that is, named virtual entities to which privileges are assigned; by activating a role, users are enabled to execute the privileges associated with the role.

Users and roles can be granted authorizations on any object managed by the DBMS, namely: tables, views, columns of tables and views, domains, assertions, and user-defined constructs such as user-defined types, triggers, and SQL-invoked routines. Authorizations can also be granted to `public`, meaning that they apply to all the user and role identifiers in the SQL environment. Apart from the `drop` and `alter` statements – which permit to delete and modify

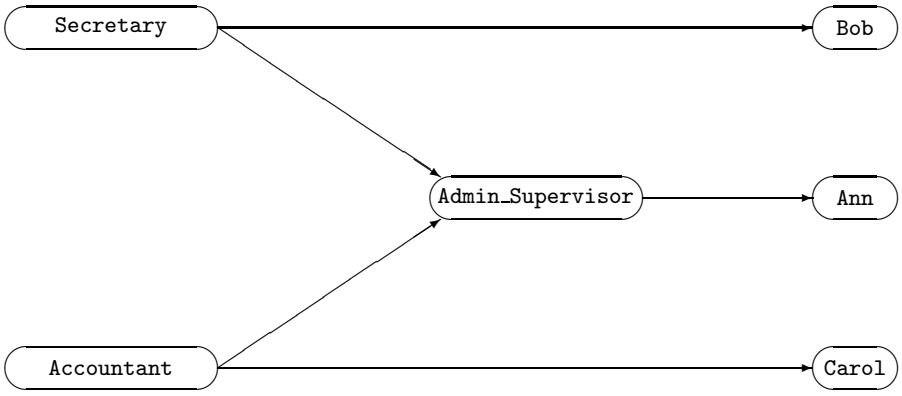

Figure 2. An example of role chains in SQL

the schema of an object and whose execution is reserved to the object's owner – authorizations can be specified for any of the commands supported by SQL, namely: *select*, *insert*, *update*, and *delete* for tables and views (where the first three can refer to specific columns), and *execute* for SQL-invoked routines. In addition, other actions allow controlling references to resources; they are: *reference*, *usage*, *under*, and *trigger*. The *reference* privilege, associated with tables or attributes within, allows referring to tables/attributes in an integrity constraint: a constraint cannot be checked unless the owner of the schema in which the constraint appears has the *reference* privilege on all the objects involved in the constraint. The reason for this is that constraints may affect the availability of the objects on which they are defined, and therefore their specification should be reserved to those explicitly authorized. The *usage* privilege, which can be applied to domains, user-defined types, character sets, collations, or translations, allows the use of the object in one's own declarations. The *under* privilege can be applied to a user-defined type and allows subjects to define a subtype of the specified type. The *trigger* privilege, referred to a table, allows the definition of a trigger on the table.

Besides authorizations to execute privileges on the different objects of the database management system, SQL also supports authorizations on roles. In particular, roles can be granted to other users and roles. Granting a role to a user means allowing the user to activate the role. Granting a role r' to another role r means permitting r to enjoy the privileges granted to r'. Intuitively, authorizations on roles granted to roles introduce chains of roles through which privileges can flow. For instance, consider the case in Figure 2 where the rightmost three nodes are users, the remaining three nodes are roles and an arc corresponds to an authorization of the incident node on the role source of the arc (e.g., `Ann` has an authorization for the `Admin_Supervisor` role). While each of the users will be allowed to activate the role for which it has the authorization (directly connected in our graph) it will enjoy the principles of all the roles rechable through a chain. For instance, when activating role `Admin_Supervisor`,

`Ann` will enjoy, besides the privileges granted to this role, also the privileges granted to roles `Secretary` and `Accountant`.

3.1.1. Access control enforcement

The subject making a request is always identified by a pair $\langle uid, rid \rangle$, where *uid* is the SQL-session user identifier (which can never be null) and *rid* is a role name, whose value is initially null. Both the user identifier and the role identifier can be changed via commands `set session authorization` and `set role`, respectively, whose successful execution depends on the specified authorizations. In particular, enabling a role requires the current user to have the authorization for the role. The current pair $\langle uid, rid \rangle$ can also change upon execution of an SQL-invoked routine, where it is set to the owner of the routine (cf. Section 3.1.3). An *authorization stack* (maintained using a "last-in, first-out" strategy) keeps track of the sequence of pairs $\langle uid, rid \rangle$ for a session. Every request is controlled against the authorizations of the top element of the stack. Although the subject is a pair, like for authorizations and ownership, access control always refers to either a user or a role identifier in mutual exclusion: it is performed against the authorizations for *uid* if the *rid* is null; it is performed against the authorizations for *rid*, otherwise. In other words, by activating a role a user can enjoy the privileges of the role while disabling her own. Moreover, at most one role at a time can be active: the setting to a new role rewrites the *rid* element to be the new role specified.

3.1.2. Administration

Every object in SQL has an owner, typically its creator (which can be set to either the `current_user` or `current_role`). The owner of an object can execute all privileges on it, or a subset of them in case of views and SQL-invoked routines (cf. Section 3.1.3). The owner is also reserved the privilege to `drop` the object and to `alter` (i.e., modify) it. Apart for the `drop` and `alter` privileges, whose execution is reserved to the object's owner, the owner can grant authorizations for any privilege on its objects, together with the ability to pass such authorizations to others (`grant option`).

A `grant` command, whose syntax is illustrated in Figure 3, allows granting new authorizations for roles (enabling their activation) or for privileges on objects. Successful execution of the command requires the grantor to be the owner of the object on which the privilege is granted, or to hold the grant option for it. The specification of `all privileges`, instead of an explicit privilege list, is equivalent to the specification of all the privileges, on the object, for which the grantor has the grant option. The `with hierarchy option` (possible only for the select privilege on tables) automatically implies granting the grantee the *select* privilege on all the (either existing or future) subtables of the table on which the privilege is granted. The `with grant option` clause (called `with admin option` for roles) allows the grantee to grant others the received authorization (as well as the grant option on it). No cycles of role grants are allowed.

The revoke statement allows revocation of (administrative or access) privileges previously granted by the revoker (which can be set to the `current_user` or `current_role`). Due to the use of the grant option, and the existence of derived objects (see Section 3.1.3), revocation

```
grant all privileges | <action>
on [ table ] | domain | collation | character set | translation | type <object name>
to <grantee> [{<comma> <grantee>}...]
[ with hierarchy option ]
[ with grant option ]
[ granted by <grantor>]
```

```
grant <role granted> [{ <comma> <role granted> }...]
TO <grantee> [{ <comma> <grantee> }...]
[with admin option]
[granted by <grantor>]
```

```
revoke [ grant option for | hierarchy option for ] <action>
on [ table ] | domain | collation | character set | translation | type <object name>
from <grantee> [ { <comma> <grantee> }... ]
[ granted by <grantor> ]
cascade | restrict
```

```
revoke [ admin option for ]
<role revoked> [ { <comma> <role revoked> }... ]
from <grantee> [ { <comma> <grantee> }... ]
[ granted by <grantor> ]
cascade | restrict
```

Figure 3. Syntax of the grant and revoke SQL statements

of a privilege can possibly have side effects, since there may be other authorizations that depend on the one being revoked. Options `cascade` and `restrict` dictate how the revocation procedure should behave in such a case: `cascade` recursively revokes all those authorizations which should not exist anymore if the requested privilege is revoked; `restrict` rejects the execution of the revoke operation if other authorizations depend on it. To illustrate, consider the case where user `Ann` creates a table and grants the select privilege, and the grant option on it, to `Bob` and `Carol`. `Bob` grants it to `David`, who grants it to `Ellen`, and to `Frank`, who grants it to `Gary`. `Carol` also grants the authorization to `Frank`. Assume for simplicity that all these grant statements include the grant option. Figure 4(a) illustrates the resulting authorizations and their dependencies via a graph reporting a node for every user and an arc from the grantor to the grantee for every authorization. Consider now a request by `Ann` to revoke the privilege from `Bob`. If the revoke is requested with option `cascade`, also the authorizations granted by `Bob` (who would not hold anymore the grant option for the privilege) will be revoked, causing the revocation of `David`'s authorization, which will recursively cause the revocation of the authorization `David` granted to `Ellen`. The resulting authorizations (and their dependencies) are as illustrated in Figure 4(b). Note that no recursive revocation is activated for `Frank` as, even if the authorization he received from `Bob` is deleted, `Frank` still holds the privilege with the grant option, (received from `Carol`). By contrast, if `Ann` were to request the revoke operation with the `restrict` option, the operation would be refused because of the authorizations dependent on it (those which would be revoked with the `cascade` option).

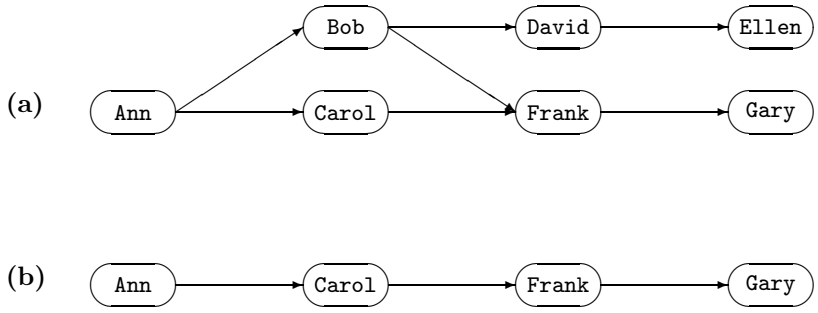

Figure 4. A graphical representation of authorizations before (a) and after (b) a cascade revocation

3.1.3. Views and invoked routines

A special consideration must be devoted to authorizations for derived objects (views) and SQL-invoked routines, where the case can be that the owner creating them does not own the underlying objects used in their definition.

A view is a *virtual* table derived from base tables and/or other views. A view definition is an SQL statement whose result defines the content of the view. The view is virtual since its content is not explicitly stored but it is derived, at the time the view is accessed, by executing the corresponding SQL statement on the underlying tables. A user/role can create a view only if it has the necessary privilege on all the views, or base tables, directly referenced by the view. The creator receives on the view the privileges that it holds on all the tables directly referenced by the view. Also, it receives the grant option for a privilege only if it has the grant option for the privilege on all the tables directly referenced by the view. If it holds a privilege on the view with the grant option, the creator can grant the privilege (and the grant option) to others. The grantees of such privileges need not hold the privileges on the underlying tables to access the view; access to the view only requires the existence of privileges on the view. Intuitively, the execution of the query computing the view is controlled against the authorizations of the view's owner (similarly to what the *suid* bit does in Unix). Also, views provide a way to enforce finer-grained access (on specific tuples). For instance, a user can define a view EU_EMPLOYEES on table EMPLOYEES containing only those rows for which the value of attribute `nationality` is equal to `EU`. She can then grant other users the *select* privilege on the view, thus allowing (and restricting) them access to information on employees within the European Union. Views are the only means to bypass an *all tuples* or *no tuple* access on tables. While convenient, views are however simply a trick for enforcing content-dependent fine-grained access control (which is not the main reason why they were developed) and as such result limiting for this purpose:

a different view should be defined for any possible content-dependent access restriction which should be enforced.

An SQL-invoked routine is an SQL-invoked procedure, or an SQL-invoked function, characterized by a header and a body. The header consists of a name and a, possibly empty, list of parameters. The body may be specified in SQL or, in the case of external routines, written in a host programming language. At object creation time, a user is designated as the *owner* of the routine. Analogously to what is required for views, to create an SQL-invoked routine, the owner needs to have the necessary privileges for the successful execution of the routine. The routine is dropped if at any time the owner loses any of the privileges necessary to execute the body of the routine. When a routine is created, the creator receives the *execute* privilege on it, with the grant option if it has the grant option for all the privileges necessary for the routine to run. If the creator of a routine has the *execute* privilege with the grant option, she can grant such a privilege, and the grant option on it, to other users/roles. The *execute* privilege on an SQL routine is sufficient for these other users/roles to run the routine (they need not have the privileges necessary for the routine to run; only the creator does). Intuitively, SQL routines provide a service similar to the *setuid*/*setgid* privileges in Linux and impersonation in Windows (controlling privileges with respect to the owner instead of the caller of a procedure).

4. Access control for Internet-based solutions

We here survey the most common security features for Internet-based solutions. Again, we illustrate the most popular representative of the different families. We will therefore look at TCPD for Internet service access, at Apache for web-based solutions, and Java 2 security model.

4.1. TCPD

The `tcpd` program (`www.porcupine.org/wietse`) is a wrapper program that is normally used in Unix-like operating systems to monitor incoming requests for Internet services such as `telnet`, `finger`, and `ftp`, among others. `tcpd` is activated by the `inetd` every time a request for service is received on a port. Upon activation, `tcpd` logs the request (recording the timestamp, the client host name, and the name of the requested service) as specified in `etc/syslog.conf` and evaluates files `/etc/hosts.allow` and `/etc/hosts.deny` (which are the files where access control rules are specified) to determine whether the request should be granted or denied. Each of these files include zero or more *access rules* of the form:

$$\textit{service\_list} : \textit{client\_list} \; [: \textit{shell\_command} \;]$$

where *service_list* is a list of service daemons (e.g., `ftpd`, `telnetd`, and `fingerd`); *client_list* is a list of host names, host addresses, or patterns; and *shell_command* is an optional shell command that must be executed every time the rule is matched. Wildcards can be used in place of a specific service/client to denote a set of them. For instance, wildcard `ALL` matches with any service/client, while `LOCAL` matches any host whose name does not contain a dot

character. Patterns are partial host/address specification and are used to refer, in a convenient way, to groups of hosts or addresses (all those matching with the pattern). Typically a pattern specifies only the most generic part of a host/address identifier (namely the rightmost elements for symbolic addresses and the leftmost elements for numeric IPs) thus denoting a whole subnetwork of machines. In other words, a symbolic pattern begins with a dot character and matches all host names whose rightmost components equal those specified. For instance, patterns `.it` or `.acme.com` will match all machines in the `it` domains or within the `acme.com` subnetwork, respectively. Conversely, a numeric pattern ends with a dot character and matches all host addresses whose leftmost fields equal those specified. For instance, pattern `159.155.` will match all machines in the `159.155.` subnetwork.

The difference between files `hosts.allow` and `hosts.deny` is that `hosts.allow` expresses permissions (i.e., which hosts should be allowed access to the mentioned services), while `hosts.deny` expresses denials.

The access control process performs first an evaluation of `hosts.allow`. If a matching rule is found access is granted (and the shell command executed, if any). Otherwise, `hosts.deny` is evaluated and, if a matching rule is found, access is denied (and the shell command executed, if any). The fact that the evaluation order is established by the mechanism implies that only this single predefined conflict resolution policy is supported. If no rule is found in either files, the access is granted (open policy by default). As an example, the following specifications:

```
#hosts.allow
  in.ftpd: ALL: mail -s "remote ftp attempt from %h" admin)

#hosts.deny
  ALL: ALL
```

deny all accesses but `ftp`. The shell command in the permission, executed in correspondence of `ftp` requests, sends an email message to the system administrator signaling the ftp request from client `%h`, where the symbolic name `%h` is expanded to the client host name or IP address.

4.2. Apache access control

The Apache HTTP server (`www.apache.org`) allows the specification of access control rules via a per-directory configuration file usually called `.htaccess` [1]. The `.htaccess` file is a text file including access control rules (called *directives* in Apache) that affect the directory in which the `.htaccess` file is placed and, recursively, directories below it (see Section 4.2.4 for more details). Figure 5 illustrates a simple example of `.htaccess` file.

Access control directives, whose specification is enabled via module `mod_access`, can be either *host-based* (they can refer to the client's host name and IP address, or other characteristics of the request) or *user-based* (they can refer to usernames and groups thereof). We start by describing such directives and then illustrate how the `.htaccess` files including them are evaluated.

```
SetEnvIf Referer www.mydomain.org internal_site
AuthName "user-based restriction"
AuthType Basic
AuthUserFile /home/mylogin/.htpasswd
AuthGroupFile /home/mylogin/.htgroup
Order Deny,Allow
Deny from all
Allow from acme.com
Allow from env=internal_site
Require valid-user
Satisfy any
<FilesMatch public_access.html>
  Allow from all
</FilesMatch>
```

Figure 5. A simple example of `.htaccess` file

4.2.1. Host-based access control

Host-based directives resemble and enrich the security specifications of the `tcpd` solution examined earlier. Both permissions and denials can be specified, by using the `Allow` (for permissions) and `Deny` (for denials) directives, which can refer to location properties or environment variables of the request. *Location-based* specifications have the form:

`Allow from` *host-or-network*`/all`
`Deny from` *host-or-network*`/all`

where *host-or-network* can be: a domain name (e.g., `acme.com`); an IP address or IP pattern (e.g., `155.50.`); a network/netmask pair (e.g., `10.0.0.0/255.0.0.0`); or a network/n CIDR mask size, where n is a number between 1 and 32 specifying the number of high-order 1 bits in the netmask. For instance, `10.0.0.0/8` is the same as `10.0.0.0/255.0.0.0`. In alternative value `all` denotes all hosts on the network.

Variable-based specifications have the form:

`Allow from env =` *env-variable*
`Deny from env =` *env-variable*

where *env-variable* denotes an environment variable. The semantics is that the directive (`allow` or `deny`) applies if *env-variable* exists. Apache permits to set environment variables based on different attributes of the HTTP client request using the directives provided by module `mod_setenvif`. The attributes may correspond to various HTTP request header fields (see RFC 2616 [7]) or to other aspects of the request. The most commonly used request header field names include: `User-Agent` (the user agent originating the request) and `Referer` (the URI of the document from which the URI in the request was obtained). For instance, in Figure 5

directive `SetEnvIf` sets "internal_site" if the referring page was in the `www.mydomain.org` Web site. The "`Allow from env =` internal_site" directive, then, permits access if the referring page matches the given URI.

Access control evaluates the content of file `.htaccess` to determine whether a request should be granted or denied. The `Order` directive controls the order in which the `Deny` and `Allow` directives must be evaluated (thus allowing users to dictate the conflict resolution policy to be applied) and defines the default access state. There are three possible orderings:

- `Deny,Allow`: the deny directives are evaluated first, and access is allowed by default (open policy). Any client that does not match a deny directive *or* matches an allow directive is granted access.
- `Allow,Deny`: the allow directives are evaluated first and access is denied by default (closed policy). Any client that does not match an allow directive *or* matches a deny directive is denied access.
- `Mutual-failure`: only clients that do not match any `Deny` directive *and* match an `Allow` directive are allowed access.

For instance, the `.htaccess` file in Figure 5 states that all hosts in the `acme.com` domain and requests with a referring page in the `www.mydomain.org` Web site are allowed access; all other hosts are denied access.

4.2.2. User-based access control

Besides host-based access control rules, Apache includes a module, called `mod_auth`, that enables user authentication (based on usernames and passwords) and enforcement of user-based access control rules. Usernames and associated passwords are stored in a text *user file*, reporting pairs of the form "*username:MD5-encrypted password*". Command `htpasswd` is used to modify the file (i.e., add new users or change passwords) as well as to create/rewrite it (a `-c` flag rewrites the file as new). The command has the form:

`htpasswd` [-c] *filename username*

where *filename* is the full path name of the user file and *username* is the name of the user we are creating. Upon entering the command, the system will ask to specify the password (as usual asking its input twice to avoid insertion errors). An alternative to the text user file provided by module `mod_auth` is given by modules `mod_auth_db` and `mod_auth_dbm`. With these modules, the usernames and passwords are stored in Berkeley DB files and DBM type database files, respectively.

To define user-based restrictions, a name can be given to the portion of the file system access to which requires authentication. This portion, called *realm*, corresponds to the subtree rooted at the directory containing the `.htaccess` file.

The main directives to create realms are:

- `AuthName`, to give a name to the realm. The realm name will be communicated to users when prompted for the login dialog (e.g., as in Figure 6).

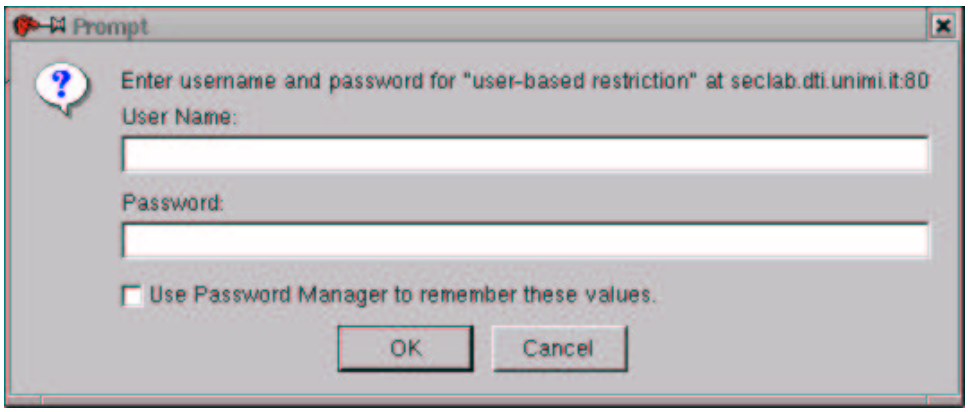

Figure 6. An example of dialog box that prompts for username and password

- `AuthType`, to specify the type of authentication to be used. The most common method, implemented by `mod_auth` is `Basic`, which sends the password from the client to the server unencrypted (with a base64-encoding). A more secure, but less common alternative is the `Digest` authentication method, implemented by module `mod_auth_digest`, which sends the server a one-way hash (MD5 digest) of the username:password pair. This Digest authentication method is supported only by relatively recent versions of browsers (e.g., Opera 4.0, MS Internet Explorer 5.0, and Amaya).
- `AuthUserFile`, to specify the absolute path of the file that contains usernames and passwords. Note that the user file containing names and passwords does not need to be in the same directory as the `.htaccess` file.
- `AuthGroupFile`, to specify the location of a *group file*, and therefore provide support for access rules specified for groups. The group file is a list of entries of the form

  group-name: username1 username2 username3 .....

  where *group-name* is the name associated with the group to which the specified usernames are declared to belong, and each of the usernames appearing in the list must be in the user file (i.e., be an existing username).

The four directives above allow the server to know where to find the usernames and passwords and what authentication protocol has to be used. User-based access rules are specified with a directive `require` that can take three forms:

- `require user` *username1 username2 ... usernameN*
  only usernames "*username1 username2 ... usernameN*" are allowed access;
- `require group` *group1 group2 ... groupM*
  only usernames in groups "*group1 group2 ... groupM*" are allowed access;

- `require valid-user`
  any username in the user file is allowed access.

4.2.3. Host-based and user-based interactions and finer-grained specifications

Host- and user-based access directives are not mutually exclusive and they can both be used to control access to the same resources. Directive `Satisfy` allows the specification of how the two sets of directives should interact. `Satisfy` takes one argument whose value can be either `all` or `any`. Value `all` requires both user-based and host-based directives to be satisfied for access to be granted, whereas for value `any`, it is sufficient that either one is satisfied for access to be granted.

As already said, all the directives specified in `.htaccess` apply to the file system subtree rooted at the directory that contains the specific `.htaccess` file unless overridden. In other words, a `.htaccess` file in a directory applies to all the files directly contained in the directory and recursively propagates to all its subdirectories unless a `.htaccess` has been specified for them (*most specific takes precedence*).

Apache 1.2 and later support finer-grained rules allowing the specification of access directives on a per-file basis by including `FilesMatch` section of the form "<`FilesMatch` *reg-exp*> *directives* </`FilesMatch`>", with the semantics that the directives included in the `FilesMatch` section apply only to the files with a name matching the regular expression specified. Also, directives can be specified on a per-method basis, by using a `Limit` section of the form "<`Limit` *list of access methods*> *directives* </`Limit`>", with the semantics that the directives included in the `Limit` section apply only to the accesses listed (again overriding the directives specified in the `.htaccess` file). As an example, directive

```
<Limit get post put>
  require valid-user
</Limit>
```

would allow any authenticated user to execute methods `get`, `post`, and `put`. The directives does not apply to other operations.

4.2.4. Evaluation of `.htaccess` *files*

As mentioned previously, file `.htaccess` is used to control accesses to the files in a directory. Therefore, whenever an access request to a file is submitted, the Apache HTTP server starts checking in the top directory for a `.htaccess` file, and then checks each subdirectory down to and including the directory that the requested file is in [5]. All files `.htaccess` finded during this process (called *directory walk*) are processed and merged thus resulting in a set of directives that apply to the requested file. More precisely, the directives specified in the `.htaccess` files have to be processed if they belong to the categories (`AuthConfig`, `FileInfo`, `Indexes`, `Limit`, and `Options`) listed in the `AllowOverride` list specified in a server configuration file. These directives are then merged according to the most specific principle, that is, directives within files `.htaccess` in subdirectories may change or nullifies the effects of the directives within files `.htaccess` of parent directories. As an example, suppose that the access request for `http://acme.com/Department1/welcome.html` resolves to the

file `/home/myaccount/www/Department1/welcome.html` and that statement `AllowOverride All` has been specified. In this case, the Apache HTTP server merges *all* directives included in the `.htaccess` files of directories: `/`; `/home`; `/home/myaccount`; `/home/myaccount/www`; and `/home/myaccount/www/Department1`.

4.3. Java 2 security model

Java is both a modern object-oriented programming language and a complex software architecture. Java has been developed by Sun Microsystems and is currently one of the most important solutions for the construction of applications in a network environment. Java offers sophisticated solutions for the design of distributed and mobile applications, where the software can be partitioned on distinct nodes and downloaded from one node to be executed on another.

Since its introduction, Java designers have carefully considered the security implications of an architecture where executable code could be downloaded from the network, possibly from untrusted hosts. The first security model of Java, the one associated with Java Development Kit version 1.0 (JDK 1.0), was based on the construction of a *sandbox*, a restricted environment for the execution of downloaded code, with rigid restrictions on the set of local resources that could be used (e.g., with no access to the file system and with limits on network access).

The main problem of JDK 1.0 security model was the limited granularity and the availability of a single policy for all downloaded code. JDK 1.0 would let programmers revise the access control services and implement their own version; but the implementation of access control services is complex, expensive and delicate, making it unfeasible for most applications.

The evolution of Java to version 2 gave the opportunity to revise the security model and significantly improve it. We describe the Java 2 Security Architecture. A full and authoritative description of the architecture appears in [8].

We observe that the security services of Java are not related with the access control system of the host operating system. This design choice derives from the requirement to make Java a fully portable execution environment, that does not depend on the services of the underlying system. The Java environment will have to be properly protected on the host system, as write access to the implementation of the Java Virtual Machine, or to its configuration, would permit to bypass any security mechanism within the Java environment.

We focus the presentation on the security model that associates permissions with pieces of Java code. This code-centric model adequately supports the security of mobile code. We do not describe the *Java Authentication and Authorization Service* (JAAS, since Java 2 v. 1.4 integrated with the JDK), a set of Java packages that offer services for user authentication and management of access control rights. JAAS extends the native Java 2 security model, using all the mechanisms presented here.

4.3.1. Security Policy

The security policy describes the behavior that a Java program should exhibit. Each security policy is composed of a list of entries (an access control list) that define the permissions associated with Java classes and applications. There is a standard security policy defined for the whole Java installation, and each user can personalize it extending the ACL in several

ways, for example, writing a specific file in the personal home directory. The security policy is represented by a `Policy` object.

Each entry in the security policy describes a piece of Java code and the permissions that are granted to it. Each piece of Java code is described by a URL and a list of signatures (represented in Java by a `CodeSource` object). The URL can be used to identify both local and remote code; with a single URL it is also possible to characterize single classes or complete collections (packages, JAR files, directory trees). The signatures may be applied on the complete URL or on a single class within a collection. Since URLs may identify collections, it is important to support implication among `CodeSource` objects (e.g., `http://www.xmlsec.org/classes/` implies `http://www.xmlsec.org/classes/xml.jar`).

4.3.2. Permissions

Permissions describe the access rights that are granted to pieces of Java code. Each permission is represented by an instance of the abstract class `Permission`. Permissions are typically represented by a target and an action (e.g., file target `/tmp/javaAppl/buffer` and action `write`). There are permissions that are characterized only by the target, with no action (e.g., target `exitVM` for the execution of `System.exit`). The `Permission` class is specialized by many concrete classes, which define a hierarchy. Direct descendants of `Permission` are `FilePermission` (used to represent access rights on files), `SocketPermission` (used to control access to network ports), `AllPermission` (used to represent with a single permission the collection of all permissions) and `BasicPermission` (typically used as the base class for permissions with no action).

The current security model considers only positive permissions. The rationale is that the evaluation is more efficient and the model is clearer for the programmer. However, no fundamental restriction has been introduced and the model could evolve to support negative authorizations (in a future version of Java, or in an ad-hoc security mechanism built for a specific application).

It is also interesting to note that permissions refer to classes and not to instance objects. A model granting permissions to objects would have offered finer granularity, but it would have also been more difficult to manage. Specifically, objects exist only at run-time, whereas the security policy is static and it is not convenient to specify in it permissions at the level of objects.

To manage sets of permission, the Java model offers class `PermissionCollection`, that groups permissions of the same category (e.g., file permissions). Class `Permissions` represents collections of `PermissionCollection` objects, that is, collections of collections of `Permission` objects.

4.3.3. Access control

In the Java 2 architecture, permissions are not directly associated with classes. Class `ProtectionDomain` realizes the link between classes and permissions. The security policy specifies permissions for a URL which may correspond to many classes; all the classes refer to

the same protection domain. There is a predefined *system* domain that associates permission `AllPermissions` to all the classes in the core of the Java architecture.

In Java 2, access control is realized at two levels: `SecurityManager` and `AccessController`. At the higher level, class `SecurityManager` is responsible for evaluating access restrictions and is invoked whenever permissions have to be verified. In JDK 1.0 the class was abstract, forcing each Java implementation to provide its own realization. In Java 2 the class is concrete and a standard implementation is part of the run-time environment.

The main method of class `SecurityManager` is `checkPermission`. In JDK 1.0 the check on permissions was realized by ad-hoc methods (e.g., to check for read permission on a file, method `checkRead` was used). Java 2 maintains all the previous methods for backward compatibility, but it uses a single method `checkPermission` for every permission type. This increases the flexibility of the security model, as the introduction of novel permissions can be managed with relative ease, without the need to modify the implementation of the `SecurityManager`.

Method `checkPermission` determines whether if the permission that appears as first parameter of the method is granted. If the check is successful, the method returns the control to the caller, otherwise it generates a security exception.

Method `checkPermission` in the standard `SecurityManager` immediately calls method `checkPermission` of class `AccessController`. Class `AccessController` is a final (i.e., unmodifiable) class that represents the security policy that Java 2 supports by default. This distinction into two levels is motivated by two conflicting requirements, each managed at a separate level. On the one hand, there is the need for flexibility, for applications that may need a different security policy; for these applications it would be possible to realize a specialized implementation of the `SecurityManager` class, that would then be automatically invoked for security checks by Java classes (that call the services of the `SecurityManager`). On the other hand, applications may prefer to have a guarantee that the security model used is the default one for Java 2; in this case, applications may opt to refer directly to the services of the `AccessController` class.

Access control is evaluated in the execution environment, which is characterized by an array of `ProtectionDomain` objects. There may be more than one `ProtectionDomain` object as Java classes may invoke the services of classes that belong to different domains. The problem is then to decide how to consider the permissions of different domains in the execution environment. The solution used in Java 2 is to consider as applicable permissions only the permissions that belong to the intersection of all the domains. Consequently, when `checkPermission` runs, it considers all the `ProtectionDomain` objects and if there is at least one domain that has not been granted the permission being checked, a security exception is generated. The rationale for this policy is that this is the safest approach, realizing the minimum privilege principle.

There is an exception to the above behavior, which requires the use of method `doPrivileged` of class `AccessController`. Method `doPrivileged` creates a separate execution environment, which considers only the permissions of the `ProtectionDomain` associated with the code itself. The goal of this method is analogous to that of the *setuid* mechanism in Linux, where the privileges of the owner of the code are granted to the user executing it. For instance, a `changePassword` method that requires write permission on a password file can be realized within a `doPrivileged` method. The advantage of this mechanism with respect to the *setuid* mechanism is that in Java it is possible to restrict with a very fine granularity the Java

statements that have to be executed in a privileged mode, whereas in Linux the privileges of the owner are available to the executor for the complete run of the program (in contrast to the least privilege).

Finally, we consider how the security model integrates with the inheritance mechanism that characterized the Java object model. Two classes where one is a specialization of the other may belong to distinct domains. When a method of a subclass is invoked, the effective `ProtectionDomain` is the one where the method is implemented; if the method is simply inherited from the superclass, with no redefinition, the domain of the superclass is considered; if the method is redefined in the subclass, the domain of the subclass is instead used by the `checkPermission` method.

5. Conclusions

In this paper we have discussed the basic concepts of access control and illustrated the main features of the access control services provided by some of the most popular operating systems, database management systems, and network-based solutions. Hinting at the principles and how they are (or are not satisfied) by current approaches, the paper can be useful to both those interested in access control development, who may get an overview of a wide array of solutions in many different contexts, and to those end users who need to represent their protection requirements in their systems and, by knowing their strengths and weaknesses, can make a more proper and secure use of it.

REFERENCES

1. Apache HTTP server version 1.3. http://httpd.apache.org/docs/.
2. P. Atzeni, S. Ceri, S. Paraboschi, and R. Torlone. *Database Systems - Concepts, Languages and Architectures*. McGraw-Hill, 1999.
3. S.M. Bellovin. Security problems in the tcp/ip protocol suite. http://www.citi.umich.edu/u/provos/security/ipext.ps.gz.
4. S. Castano, M.G. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison-Wesley, 1995.
5. K. Coar. Using .htaccess files with apache, 2000. http://apache-server.com/tutorials/ATusing-htaccess.html.
6. Database Language SQL – Parts 1–5. ISO International Standard, ISO/IEC 9075:1999, 1999.
7. R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. *Hypertext Transfer Protocol – HTTP/1.1*, June 1999. http://www.rfc-editor.org/rfc/rfc2616.txt.
8. L. Gong. *Inside Java 2 Platform Security*. Addison-Wesley, 1999.
9. S. Jajodia, P. Samarati, M.L. Sapino, and V.S. Subrahmanian. Flexible support for multiple access control policies. *ACM Transactions on Database Systems*, 26(2):18–28, June 2001.
10. P. Samarati and S. De Capitani di Vimercati. Access control: Policies, models, and mechanisms. In R. Focardi and R. Gorrieri, editors, *Foundations of Security Analysis and Design*, LNCS 2171. Springer-Verlag, 2001.
11. R. Sandhu. Separation of duties in computerized information systems. In *Proc. of the IFIP WG11.3 Workshop on Database Security*, Halifax, U.K., September 1990.
12. R. Sandhu, D. Ferraiolo, and R. Kuhn. The NIST model for role-based access control: Towards a unified standard. In *Proc. of the fifth ACM Workshop on Role-based Access Control*, pages 47–63, Berlin Germany, July 2000.
13. R. Sandhu and P. Samarati. *CRC Handbook of Computer Science and Engineering*, chapter Authentication, access control and intrusion detection, pages 1929–1948. CRC Press Inc., 1997.

14. R.S. Sandhu, E.J. Coyne, H.L. Feinstein, and C.E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, February 1996.