A Survey of Access Control Policies

Amanda Crowell
University of Maryland
acrowell@cs.umd.edu

ABSTRACT
Modern operating systems each have different implementations of access controls and use different policies for determining the access that subjects may have on objects. This paper reviews the three main types of access control policies: discretionary, mandatory, and role-based. It then describes how Windows and various Unix systems implement their chosen access control policies. Finally, it discusses some common mistakes that application programmers make when enforcing access control in their applications and gives some general guidance to reduce the occurrence of mistakes.

1. INTRODUCTION
Modern operating systems use different mechanisms for providing access control. These mechanisms are designed to meet security policies that vary in the way access decisions are made. In all the systems, there are two major components: the *subjects* who take actions and the *objects* they take actions on. Access control policies specify the conditions under which a subject is allowed to take an action on an object. Discretionary policies make decisions based on the rights that each object specifies for each subject. Mandatory polices make decisions based on the security levels of the subject and object. Role-based policies make decisions based on the roles that subjects have in the system and which roles are allowed to access which objects.

With so many options for implementing access controls, application developers must have a thorough understanding of their target system's implementation so that they can develop secure systems. The Windows access control system is largely a discretionary system, with some mechanisms for mimicking roles. The basic Unix protections are also discretionary, but are limited to only representing three levels of access: one for the object owner, one for the owner's group, and one for everyone else. Extensions to this basic control have been implemented, including POSIX.1e, Data and Type Enforcement (DTE), and Security Enhanced Linux (SELinux), all of which focus on adding mandatory mechanisms to the standard checks.

The Common Weakness Enumeration (CWE) describes several mistakes that application developers have made in regard to access control. Often, developers do not give the application objects the appropriate rights to begin with (i.e. they specify more rights than are necessary) and do not modify rights throughout the application's lifetime. Since spawned processes typically take on the rights of the spawning process, rights might be passed on to the spawned process that were unintentional. In response to these mistakes, some researchers have developed guidelines for using access controls appropriately. These include: (a) understanding what information is available to the system and how it can be used to make decisions, (b) determining exactly which rights are absolutely necessary and only allow those and (c) understand how ACLs are managed, apply to privileged users, and handle contradictory rights.

2. OVERVIEW OF ACCESS CONTROL
Access control in operating systems and applications determines how data and resources on systems can be accessed and by whom. The implementation of access control varies from simple to extremely complex. In addition, access control mechanisms are only effective when they are configured and used properly. Therefore, application developers and system administrators must have a thorough understanding of how the access control works in their systems so that data and resources are protected.

2.1 Components
Sandhu and Samarati [17] and Harrison et al [11] present the main components of an access control mechanism, and this section will present their descriptions. The two main players in a system are subjects and objects. A *subject* is an action component. It can either be a user, process, thread, or program that wishes to take certain actions in a system. An *object* is the target of certain actions by subjects. Objects are typically defined as being items in the system that a user can explicitly modify, create, and use. Some examples include files, printers, and even other subjects. Actions that a subject can perform in a system such as shutting it down and changing the system time are considered separately, since they are not actions to objects. However, the system itself could be considered an object and access control must restrict a subject's ability to perform these tasks as they would with objects.

Subject actions are defined by *access rights* that specify for each object the actions that a subject may perform. The most basic rights are *read*, *write*, *execute*, and *own*, with different types of objects having different types of rights. For example, text files can be read from and written to, but cannot be executed while program files can. Subject *ownership* of an object means that the subject should be able to control what other subjects may do with that object. In general, a system's *safety* is defined as its ability to help the user protect their objects from misuse at all times. The user should be able to specify who and what is able to access their data and the system should enforce their controls.

For a given system, an *access matrix* conceptually represents all the access rights in the system by having a row for every subject and a column for every object. The rights that a subject S has for a particular object O are placed in the cell of the matrix represented by $[S, O]$. Because a matrix would have such large storage requirements, in actual implementations of access controls either just the rows or just the columns are used to represent rights.

The row view of the matrix forms a *capability list* that shows for every subject S the rights they have for every object O in the system. In contrast, the column view of the matrix forms an *access list* that shows for every object O the subjects S who have access rights to it. The capability list makes it easy to determine all the accesses a particular subject has in a system but at the same time makes it difficult to determine, for any one object, all the subjects who have access. The access list has the opposite effect: it is easy to determine the subjects who can access an object, but not easy to determine all the objects a particular subject can access.

Regardless of the view of the access matrix chosen, the rights in a system are stored in an *authorization database*. While this database could be stored in a separate location, most often it is stored with the objects. A *reference monitor* uses this database to determine, for each attempted access by a subject, if the subject has the necessary rights to access the object. The authorization database and reference monitor are *mechanisms* of access control, and vary from system to system depending on the hardware and software available.

A system's *configuration* consists of the triple (S, O, P) where S and O are the subjects and objects in the system, respectively, and P is the conceptual access matrix. A command can *leak* rights in a configuration if, when it is run, it modifies a right in a cell of the matrix. For a single system with a single subject, the method of configuring the system is not all that important. However, as the number of subjects and objects grows, the configuration becomes much more complicated. In this scenario, a *security administrator* is needed to maintain the state of the authorization database. The administrator uses the *policies*, or high-level guidelines defined by the organization, company, etc, to control access decisions.

2.2 Access Policies

Three main types of access control policies have been explored in literature: *discretionary*, *mandatory*, and *role-based*. This section will give an overview of these policies as presented by Sandhu and Samarati [17].

2.2.1 Discretionary

Discretionary Access Controls (DAC) control access to objects based solely on the subject's identity and the rights specified for that identity on each object. In this type of system, an access control list would be used, as described before, which would specify, for each object, which subjects are allowed to access it and how. In such a system, the default level of access is $NONE$, so that access is prohibited unless it is explicitly defined. However, this system has a weakness in that it does not control the dissemination of information once it has been accessed. A subject could release information to other subjects who have not been authorized to receive it. They would do this by reading information from a file that the third party doesn't have access to and writing that information into a file the third party does have access to.

Consider a system with two users, *Alice* and *Bob*, and three objects, $file1$, $file2$ and $process1$. In a DAC system, each object would specify the rights that Alice and Bob have for them. So the ACLs for these objects might look like the following:

Object	Rights
file1	deny Alice read,write allow Everyone read,write
file2	allow Everyone read,write
process1	deny Bob execute allow Alice execute

Table 1: Example DAC List Entries

These rights can be interpreted as follows. For $file1$, everyone is allowed to read and write to the file except for *Alice*. For $file2$, everyone in the system is allowed to read and write to the file. Finally, for $process1$, *Bob* will not be allowed to execute the file while *Alice* will. If another user is added to the system, since a right is not expressly given for everyone else, the new user would get the default access, as is defined as by the system.

2.2.2 Mandatory

Unlike discretionary access control where the owner defines the access, *Mandatory Access Control (MAC)* is defined by the system policy. In a MAC system, subjects and objects are classified as having a certain security level. For an object, the security level represents the sensitivity of the information that it holds and the amount of damage that would result if the information was disclosed. For a subject, the security level represents the level of trust in the subject that it will not disclose information unless authorized. When access control decisions have to be made, the security levels of the subject(s) and object(s) in question are compared, and if the relationship between them meets the requirements of the system, then the access is granted. The following basic principles are required to hold in such a system:

- *Read Down* A subject's security level must be at least as high as the security level of the object it wishes to read. This requirement prevents a user from reading information for which it has not been trusted.

- *Write Up* A subject can only write to objects whose security levels are at the same level or higher. This requirement prevents a user from leaking information from high to low security levels.

With these basic principles, MAC is able control the dissemination of information once it has been accessed. A subject who has access to a file would not be able to write information to a file at a lower security level, so they can not give the information to a third party subject without the necessary security level to read the original file. MAC was created with the military in mind to protect information at various hierarchical classifications and with orthogonal compartments. Hierarchical levels include *UNCLASSIFIED*, *CONFIDENTIAL*, *SECRET*, and *TOP SECRET* and orthogonal compartments could include *Personally Identifiable Information (PII)*. However, a system that truly followed these requirements would be extremely rigid and difficult to both implement and use.

Again, consider the system with users *Alice* and *Bob* and the same three objects as before. In this system, the hierarchical and orthogonal security levels are the ones listed for multi level security (MLS) as defined for government classifications above. These hierarchical levels are, in order from lowest security level to highest: unclassified U, confidential C, secret S, and top secret TS and an orthogonal level is personally identifiable information PII. In this system, each subject and object has a security level, and the next table shows an example configuration.

Alice	S, PII
Bob	C
file1	U, PII
file2	TS
process1	TS

Table 2: Example MAC Entries

In this system, *Alice* can read $file1$ but *Bob* and $Process1$ cannot since they do not have the necessary compartment. *Alice* and *Bob* can only write to $file2$. $Process1$ can read and write to $file2$.

2.2.3 Role-Based

In *Role-Based Access Control (RBAC)*, subjects are assigned to roles that line up with roles that users hold in real life. A role could represent a set of actions and responsibilities that a subject has in their job. As an example, consider four types of subjects: students, teachers, principals and janitors in a school system. The student has the least amount of access and is limited to his/her desk in his classroom. The teacher has more access, including his/her desk, students' desks, his classroom, and the teacher's lounge. The principal has the most access, and has access to his/her desk, students' desks, teachers' desks, all classrooms, as well as the teacher's lounge. The janitor has orthogonal (but different) access than the principal in that he/she has access to all the physical rooms in the school but not to the desks. In a RBAC system, three roles would be created that represent these levels of access. Then, subjects in the system would be placed in their appropriate roles and each object authorizes access based on these roles.

This type of system has many benefits. Management of authorizations is simplified since it is broken into the two separate tasks of assigning rights to roles, then assigning subjects to roles. It also allows for hierarchical roles to be created, which can be seen in the example above. Once the student role is defined, the teacher role can be defined to be the student role plus the extra rights granted to teachers. Similarly, for the principal, the role can be defined to be the combination of the student and teacher roles, along with the extra rights granted to the teachers. This further simplifies management. In addition, this system would allow subjects to follow the principle of *least privilege* which states that subjects should only use the least amount of access/privilege necessary to complete a task. In RBAC, a user could take on the lowest role that has the necessary accesses to complete a task, and only go to higher roles when absolutely necessary.

As an example, consider the system from the previous sections with the addition of two roles: *teacher* and *student*. Access rights in this system would look like the following:

Alice	teacher
Bob	student
file1	teacher, student : read, write
file2	teacher: read, write
process1	teacher, student : execute

Table 3: Example RBAC Entries

In this example, $file1$ allows both the *teacher* and *student* roles to read and write to it, while $file2$ only allows the *teacher* role access. Also, both roles are able to execute $process1$.

3. ACCESS CONTROL IN MODERN OPERATING SYSTEMS

While the basic functionality of access control is well defined, it can be implemented in many different ways. Each operating system on the market today has different implementations, and this section will describe some of the differences.

3.1 Microsoft Windows

Microsoft first created NT to be comparable to Unix systems, and NT was the first 32 bit operating system they created [5]. Each major release of NT usually indicates significant changes in features and functionality of the OS product. This section will explore how the access control mechanisms evolved through each version of NT[1].

3.1.1 Windows NT

All versions of Windows use the discretionary method of access control, in the form of *discretionary access control lists (DACLs)* which are stored for each object in the system. A

[1] *Windows NT* is the product name for the operating system released using NT version 4.0. Subsequent OS releases were newer versions of NT (5.0+) and were given different product names.

DACL consists of *access control entries (ACEs)* that specify the levels of access that users have on the object. In Windows NT, there are two types of ACEs: access-allowed and access-denied. The ACE has three parts: a field specifying which type of ACE it is, a field with a *security identifier (SID)* that represents a user, and a 16-bit *access mask* that specifies the rights. The DACL, along with the owner, group, and auditing information, make up an object's *security descriptor*. Some access right inheritance occurs automatically. For example, when an object is created in a container such as a folder, some of the entries from the container's ACE are inherited into the newly created object. Figure 1 shows an ACE in NT. [8, 19, 4]

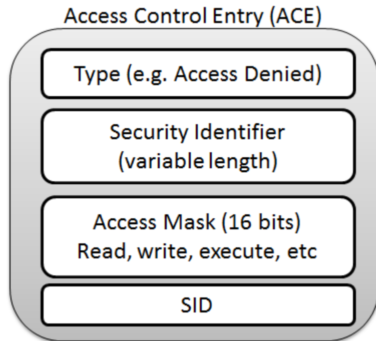


Figure 1: Access Control Entries in Windows NT

As mentioned previously, the subjects in Windows are identified by a variable length security identifier. Subjects in Windows can be users or groups of users. The information about a subject's identities and privileges is encapsulated in an *access token* that is created at logon time. This access token is used by the *Security Reference Monitor (SRM)*, a kernel component that is responsible for making access decisions. An application requesting access to an object on behalf of a subject will call the *AccessCheck* function, which takes as input the user's access token, the requested level of access, and the object's DACL. The *AccessCheck* function evaluates each ACE in the order they appear, comparing the SIDs in the ACEs to the subject's access token. All denying ACEs appear at the beginning of the ACL, and once a right has been denied it cannot be granted, even if an allow ACE occurs later in the ACL. [19, 4]

Figure 2 shows an example of an access check in Windows. Both Thread A and Thread B are requesting full access (read, write, and execute) to the object. Thread A is denied access because it contains the user "Andrew," who has been denied access by ACE1. ACE2 and 3 are not evaluated since deny ACEs take precedence over allow ACEs. Thread B is granted access since ACE1 doesn't apply, ACE2 grants write access, and ACE3 grants read and execute access.

The subject's access token is generated by the *Local Security Authority (LSA)*. This process (lsass.exe) is highly privileged and runs on each individual system. The following describes the function of the LSA and how access tokens are used:

1. User logs in. The LSA performs the authentication of the user by verifying the supplied password is correct.

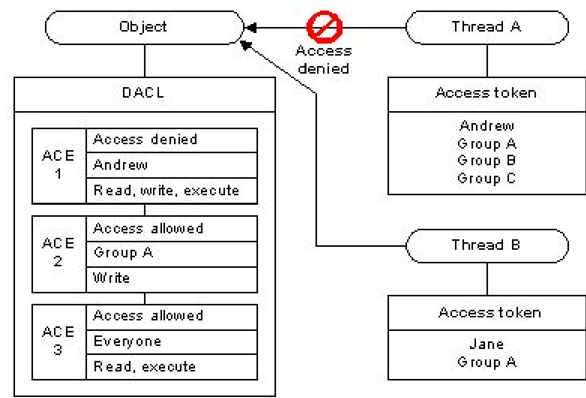


Figure 2: Example of Access Checks in Windows [4]

2. After authentication is complete, the LSA creates a logon session for the user and the access token.

3. The logon process then creates the main Windows shell (explorer.exe). The access token for the user is attached to this shell.

4. From this point on, any time that a new process is started, the user's access token gets inherited from this main Windows shell into the new process. Once a process has an access token, it cannot be replaced. However, threads may have different access tokens than their parent processes, which enables programs to execute certain functionality with different identities.

In addition to access rights to objects, user accounts (and user access tokens) can have *privileges*. Privileges are 64-bit numbers that specify access rights for actions that do not map to a specific object. They can be used to grant administrative access to large sets of objects (like system drivers) and to grant execution of operations like shutting down the system or changing the system clock. [19, 4]

Windows NT includes some built in accounts and account groups to help simplify the administration of machines. The *Administrator* account is a user accessible account that has full privilege and access rights to the system. Users can use this account to manage the other accounts and groups of the system. The *Local System* account also has complete privilege and access rights to the system. The user mode components of the operating system run under this account.

3.1.2 Windows XP / Windows 2000 / Windows Server 2003

These Windows OS versions are all based on NT version 5. Windows 2000 is 5.0, Windows XP is 5.1 or 5.2, and Windows Server 2003 is 5.2. Since all these systems are based on NT version 5.0, they are very similar in the way they perform access control.

Swift et al [19] describe some of the limitations of the access control mechanisms in Windows NT (version 4.0). One limitation is that only 16 bits are used for the access masks that specify rights in the ACEs. Because of this, only 16

access types can be specified. A second limitation is that the inheritance of access rights is not fine-grained enough to distinguish between different types of objects with different types of access. A third limitation is that when making access control changes to a tree of objects, the merging of objects with differing and already existing ACLs is ambiguous. The effect of the second and third limitations is that the result of merging and inheritance is not deterministic, and administrators (and users) might not understand how to use these functions appropriately to achieve the desired access rights on objects. A final limitation is that the only way to restrict rights of processes is to disable privileges individually, which is difficult to manage.

To alleviate these issues, NT version 5.X changes several aspects of the access control infrastructure. The first major change is to the ACEs to allow the ACE to track what type of object it is referring to. To do this, two fields were added to ACEs: *ObjectType*, which identifies the type of the object, and *InheritedObjectType*, which is used during inheritance and specifies which types of objects will inherit the ACE. Also, application developers have the option of creating their own object types and properties in addition to those that are included with the operating system. This feature allows for more customized access control. The result of adding personalized objects is the addition of new ACE types. With this addition, the following types of ACEs exist: *generic access allowed and denied* from NT 4.0, *object-specific access allowed and denied* which contain the additional information about the type of object as described above, and various callback types for use in COM systems. Figure 3 shows the structure of a generic ACE for built-in objects and Figure 4 shows an ACE for personalized object types. [6, 19, 16].

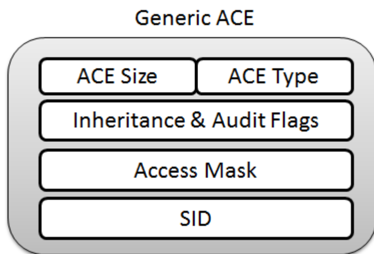


Figure 3: Generic ACE in Windows 2000

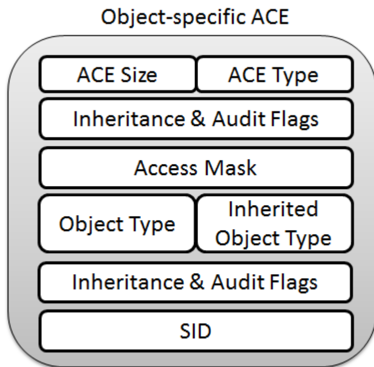
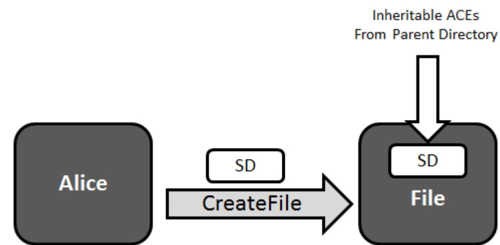
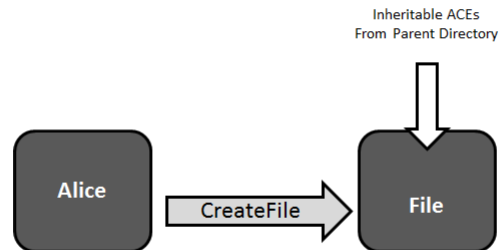


Figure 4: Object-specific ACE in Windows 2000

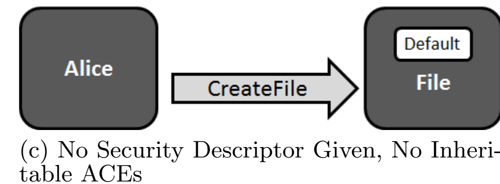
The second major change to ACEs is explicitly specifying how ACEs are inherited, both at object creation and throughout the object's lifetime. As mentioned before, in NT the inheritance specification ability was extremely limited and resulted in ambiguous inheritance. In NT 5.X, directory ACEs that should be inherited by objects created inside the directory are marked with the $OBJECT_INHERIT$ property. Similarly, directories that are created inside a directory (or container) will inherit the parent's directory ACEs that have the $CONTAINER_INHERIT$ property set. Figure 5 shows how ACEs are inherited in various scenarios of object creation. In Figure 5(a), the caller, Alice in this case, creates an object (file) and provides a security descriptor (SD). The new object will use the SD given and merge any inheritable ACEs from the container (directory in which the file was created) into the DACL in the SD. In Figure 5(b), Alice does not provide a SD, so the file just inherits ACEs from the directory to fill the DACL in its SD. In Figure 5(c), Alice does not provide a SD and there are no inheritable ACEs in the directory, so the file gets the default DACL provided by the system. And finally, in Figure 5(d) the system does not have a default DACL. So since Alice did not provide one and nothing was inherited, the file gets no DACL. This means that all users will have full access to the file. [16]



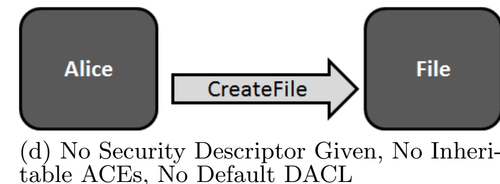
(a) Security Descriptor Given



(b) No Security Descriptor Given



(c) No Security Descriptor Given, No Inheritable ACEs



(d) No Security Descriptor Given, No Inheritable ACEs, No Default DACL

Figure 5: ACE Inheritance in Windows 2000

Rights can be changed once an object or container has been created. One way that rights can change is by changing the rights on a directory and specifying that the rights should be propagated throughout all the objects and subdirectories. When this happens, the rights should be changed in a deterministic fashion unlike in NT 4.0. It is especially important that locally specified ACEs, for example those that protect objects with private information, do not get overridden with ACEs from higher up in the hierarchy. To fix these issues, each ACE is annotated with a flag specifying whether or not it was inherited. Before ACEs are reapplied to a directory, the ACEs on objects that were inherited are removed so that local ACEs are not overwritten. In addition, all local ACEs are placed first in the DACL so that they will override any inherited ones. [16]

Two algorithms can be used to determine what level of access a subject will be granted to an object. The first algorithm calculates the maximum allowed access that the object has been given for the subject when the subject wishes to have the maximum access. It works as follows:

- If the object has no DACL, then all subjects have all access.
- If the subject/caller has the take-ownership privilege or is the owner, then algorithm grants the write owner access and read control/write-DACL accesses, respectively.
- When a deny ACE is encountered for the subject/-caller, the right is removed from the access mask that is being computed. Likewise, if an allow ACE is encountered and it has not been denied, the right is added to the access mask.
- Finally, the access mask is returned.

The second algorithm is used to check if a specific access is allowed. The algorithm works as follows:

- Again, if the object has no DACL, then the subject is granted access.
- Again, if the subject has take-ownership privilege or is the owner, the system grants access as in the first algorithm. This time, this access is used to compare to the requested rights to the ACEs in the DACL.
- Then, each ACE in the DACL is examined from first to last. An ACE is only processed if the SID in it matches the SID in the subject's access token and it is not marked as an inherit-only ACE. If an access allowed ACE is encountered matching the rights requested, then they are granted; if all requested rights have been granted then the access check is complete and the algorithm returns. If an access deny ACE is encountered that matches a requested access, then the access is denied. If the end of the DACL has been reached and access has not been granted, then access is denied. [16]

All of these changes regarding ACEs, special objects, and inheritance are part of the *Active Directory* service and functions. The goal of Active directory is to simplify the management of subject and objects throughout a domain (network) of computers. It allows domain administrators to manage all the users, groups, and object accesses from a central location and ensures that all the management settings get propagated to all computers in the domain. Some accounts are included with pre-specified levels of access that help minimize the amount of work administrators must do. Some of these accounts are as follows:

- *Local System* The most privileged account that all the user mode components of the kernel run in and has complete access to all the resources of the machine.
- *Local Service/Network Service* Does not have complete access to the machine's resources and is used to run the parts of the OS that do not need the complete access of Local System.
- *Administrator* This is the most privileged account that a user can actually log in to. From this account users can manage all accounts, including Local System, Local Service, and Network Service.

In addition, there are some default groups with specified meanings and rights that administrators can easily use and add users to. These groups include *Administrators, Authenticated Users, Everyone, Server Operators, Power Users, and Network Configuration Operators*. System administrators must decide which users get placed in which groups and in addition which groups are allowed to access which objects. Users in the Administrators group will have unlimited access to the entire system, so admission to this group should be strictly managed. The Server Operators, Power Users, and Network Configuration Operator groups will by default have access to the objects necessary to perform their roles, so users should only be placed in this group if they are trusted to perform these functions. Every user on the system gets placed in the Everyone group so administrators and normal users should be careful when specifying that the Everyone group has access[19, 8].

NT version 5.X introduced *restricted contexts*, in addition to changing the ACEs and object inheritance, which allow the administrator to specify restrictions that will force a program to run in a limited capacity. The administrator can create an access token with disabled groups and privileges, as well as restricted security identifiers. They can then run the program with that restricted SID. [16]

3.1.3 Windows Server 2008 / Windows Vista

Windows Server 2008 and Windows Vista, and even Windows 7 and Server 2008 R2, are all based on Windows NT version 6. Server 2008 and Vista are 6.0, and 7 and Server 2008 R2 are both 6.1. The basic access control mechanisms were updated from NT 5.X, but in between 6.0 and 6.1 the only real changes are in more high level protection mechanisms and functionality.

The biggest addition to access control mechanisms in Window NT 6.X was the idea of integrity levels of processes and

objects. It became important to be able to differentiate between a user and a process that user wants to run, especially when a process has been downloaded from the Internet and should not be fully trusted. Processes have an integrity level associated with them that is stored in the process token. Processes typically run with the rights of the user and could have access to the user's data. With integrity levels, user and system data can be marked with one integrity level and untrusted or high risk code (such as an Internet browser) can run at a lower integrity level. Integrity checks are performed in addition to normal access checks that prevent the lower integrity level process from crossing an integrity boundary, or modifying the user data at higher integrity levels.

With these new integrity check mechanisms, Windows was able to implement features like User Account Control (UAC) and Protected Mode Internet Explorer (PMIE). UAC warns users (by a popup) when they are executing programs/processes that try to cross integrity boundaries. PMIE warns users when webpages try to install or run programs outside of the protected Internet Explorer process. Five integrity levels were introduced and given defined uses. Table 4 describes these levels. [7, 15]

0	untrusted	most limited and blocks most write access
1	low	used by PMIE and blocks write access to most objects on the system
2	medium	basic integrity level used by normal applications launched when UAC is enabled
3	high	used by administrative applications when UAC is enabled or used by normal applications when UAC is disabled and user is administrator
4	system	used by services and other system-level applications

Table 4: Integrity levels introduced in Windows Vista/Server 2008

Integrity levels have inheritance rules similar to object ACE inheritance. In general, a process will inherit the integrity level of the process that spawned it (its parent). However, if the integrity level of the file object of the executable image and the parent's integrity level differ, the child will inherit the lowest integrity. In addition, parent processes can specify explicit integrity levels that their children will run. [15]

Objects have an integrity level defined in their security descriptor called a mandatory label. If an integrity level is not specified for an object, and the object is created by a low or untrusted integrity level process, then it will inherit an integrity level equal to the level of the process. If it was created by a medium or higher process, it will be given an integrity level of medium. When integrity level checks are performed, some policies are followed to protect objects. One policy is the *no write up* policy, which means that a lower integrity process cannot change a higher integrity object. Another policy is the *no read up* policy that is imposed on process objects and keeps a lower integrity process from reading objects at a higher integrity level. Figure 6 shows

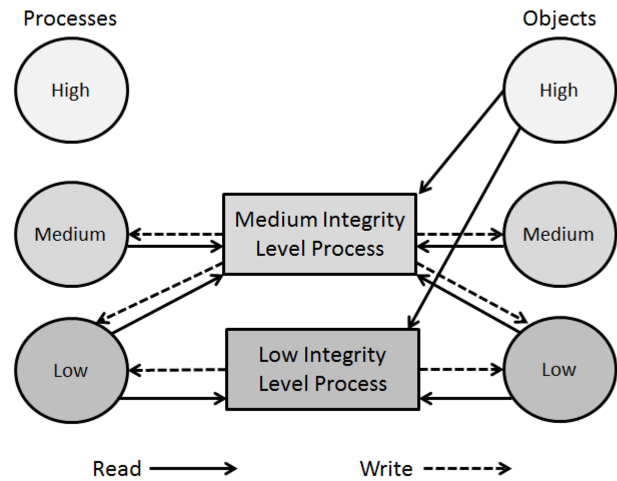


Figure 6: Medium and Low integrity Level Process Access to Other Processes and Objects [15]

some of the relationships between process and objects with medium and low integrity levels. This integrity check occurs before the discretionary access checks that were described in Section 3.1.2. [15]

3.2 Unix

Unix operating systems, including Ubuntu, RedHat, and Solaris, have very basic access control mechanisms. Each file object in the system has a trio of permissions defined by three classes of users: the owner of the file, the group the owner belongs to, and everyone else in the system. Each file has three types of rights: read (`r`), write (`w`), and execute (`x`), and these rights are separately defined for each class of users. The result is that each object has three triplets representing the rights (`rwxrwxrwx`) for that object. The first triplet represents the owner's rights, the second represents the group's rights, and the third represents the rights of everyone else. This set of rights is often referred to as the *owner-group-world* permission for the file. If a right is granted to a class of users, the bit representing the right is 1. However, if the right is not granted, then the bit is 0. For example, consider the permissions `rwxr-xr-x`. In this example the owner has read, write, and execute rights, and the group and everyone else has read and execute rights only. [2]

Bishop points out that with only three sets of permissions, this method of access control loses granularity[2]. He gives an example of a system with five users where the owner of a file wants to give each user a different set of rights to the file. There is no way to accomplish this using the basic Unix permissions as the user can only have three possibilities for the other users: in his group, not in his group, and everyone else.

Loscocco and Smalley [13] and Mayer et al [14] also point out some limitations of this simple discretionary access control policy. In general, the big issue is that a DAC model does not differentiate between human users and computer programs. So if a user starts a process, that process will run with all

the rights of the user whether it should be trusted to or not. Therefore, the DAC policy ignores other security-relevant information that could be used to make better access decisions. Some examples of these advanced decision making schemes include: taking the role of the user into account when deciding access, using the trust level of the program to assign varying levels of access, and changing access based on the sensitivity and/or integrity of the data.

3.2.1 Modifications

Because of these limitations, much work has been done to augment the basic Unix access model with more advanced access models, particularlymandatory and role-based mechanisms. Since the Linux kernel is open source, researchers and developers are able to modify it to meet their needs. Several different approaches were explored, including the POSIX.1e, Domain and Type Enforcement (DTE), and Security Enhanced Linux (SELinux) implementations. These implementations were enhanced with the development of the Linux Security Model, which sets up the framework for interacting with the kernel to make access decisions. This section will discuss these approaches in more detail.

POSIX.1e.

POSIX.1e [9] was an attempt to add many extensions to the POSIX.1 family of standards, including adding access control lists, audit mechanisms, capabilities, mandatory access control, and information labeling. While the items were not actually standardized, the work towards the standardization was released to the public, and several mechanisms have been implemented by Unix systems. The ACL is one item that was heavily adopted and implemented in many systems. Specific implementation in the various systems is not discussed since it varies from system to system.

POSIX.1e adds an *extended* ACL to what it calls the *minimal* ACL, or the standard file mode permission bits as described above (owner-group-world). In this extended model, the group class is allowed to contain ACL entries with different permission sets. In other words, the ACL can contain entries for other users and groups and get around the limitations described before of not being able to represent more than three types of users. Since the group class now represents other relationships in addition to the owning group permissions, the meaning of the group triplet in the owner-group-world permission label is changed to represent an upper bound on the permissions that any entry in the group class can obtain. In addition to modifying the file system object ACL, another ACL is defined: *default*. A default ACL is defined for a directory and specifies the permissions that files will inherit if they are created in that directory.

Access checks are performed whenever a process tries to access an object and occurs in two steps. The first step looks for the ACL entry that most closely matches the process. The second step determines if the matching entry has the necessary privileges for the access. ACL entries are examined in order of most restrictive to least restrictive, starting with the owner, then to named users, then to groups, then everyone else. While only a single entry will determine if access is granted, if a process is a member of multiple groups, each group entry will be examined. If even one of them contains the appropriate permissions, then the access will be granted.

Domain and Type Enforcement (DTE).

The goal of Domain and Type Enforcement [10] is to protect a system from attackers who subvert processes with superuser access by using mandatory access controls in addition to the standard UNIX controls. Processes are grouped into *domains*, and files are grouped into *types*. Domains can have the right to send signals and to transition into other domains. The rights to types include read, write, execute, create, and directory descend. Access decisions restrict domains to access only objects they have been specifically granted permission and also restrict domains from transitioning into other domains without explicit permission. A process can switch domains by executing a file that is an *entry point* to the other domain. Three types of domain transitions are described in Table 5.

auto	the process is automatically switched to Domain B
exec	Domain A can choose to switch to Domain B
none	Domain A cannot transition to Domain B

Table 5: Domain Transitions in DTE

If domain A has *auto* transition access to domain B, then when they execute the entry point for domain B they will automatically be transitioned into domain B. If domain A has *exec* transition access to domain B, then it can choose if it will switch to domain B. However, if domain A has the *none* transition access to domain B, then domain A can never transition to domain B and executing the entry point will not do anything.

DTE was implemented as a prototype in the 2.3.28 Linux kernel. The type information for objects is attached to the virtual file system inodes, while the domain information is attached to the process descriptor structs. A DTE policy file (`/etc/dte.conf`) contains the information about the types and domains as specified by policy administrators and is read at boot time to create the structs for each domain and an array containing all the types. A domain structure contains the types it is allowed to access, the transitions it can make to domains, its signal access to other domains, and its entry points. A file inode contains three pointers into the array of types: the *etype*, which specifies the type of the object, the *rtype*, which specifies the type of the current directory and its children, and the *utype*, which specifies the type of the object's children. An object's type will be the value of the *etype* if it has been previously determined. If the type has not yet been determined, then there are two options: either a rule exists in the policy that specifies the *rtype* for the directory or the type is inherited from the object's parent's *utype*. An access check is performed whenever a process performs an open system call. The modified kernel checks the domain structure for the current process to determine if the domain is allowed to access the type of the file. If DTE grants access, then the standard UNIX permissions

are checked.

Evolution of SELinux.

SELinux stems from an initiative by the National Security Agency (NSA) to implement a mandatory access control system that does not suffer from the limitations that accompanies these systems (rigid and difficult to implement). With the Secure Computing Corporation (SCC), NSA sought to design a system that would be flexible enough to be used with any security policy and would be acceptable for mainstream operating systems. After the architecture was developed and prototypes were tested, NSA and SCC worked with the University of Utah's Flux research group to develop the Flask architecture. This architecture was implemented in the university's Fluke operating system and used to understand and develop better support for dynamic security policies. [13]

Eventually, NSA integrated the Flask architecture into the Linux kernel and released it for use and research. Once SELinux had been presented, the consensus was that the Linux kernel needed to have a general and abstract mechanism for which access policies could be implemented. This need started the Linux Security Model (LSM) Project, which developed a kernel patch and module loading capabilities that developers can use to develop their own security policies. The model provides the basic mechanisms for attaching security fields to kernel objects and mediating access to the kernel objects, but leaves it to the policy developers to define the labels for the objects as well as what checks are performed when mediating access to objects. The LSM allows SELinux to be more easily integrated into the Linux kernel since it abstracts away the low level details. In addition, POSIX.1e and DTE have been modified to use the LSM in their current implementations. [20, 13]

The descriptions of SELinux and the Flask architecture are closely related. SELinux can be thought of as an example security policy using the Flask architecture. This section will follow the format of Smalley and Losocco [13] and present the general Flask architecture along with the specific SELinux implementation, but using the implementation for the LSM [18], not the original SELinux hand-made patches. The SELinux security policy is a mandatory access control policy that combines the ideas of type enforcement, role based access control and multi-level security (optional). It is also shipped and "on" by default in the Fedora and Red Hat operating systems.

Description of Flask/SELinux.

As mentioned previously, the Flask architecture provides the necessary mechanisms to develop an access control policy without tying the architecture to any one policy structure. Similar to other MAC systems, the objective is to label every subject and object and to mediate access from subjects have to objects and between themselves by comparing values in the labels. Flask achieves flexibility in the system by separating the portion of the system that contains the policy logic for labeling objects and determining access from the portion of the system that provides enforcement mechanisms. This idea carried over into the implementation of the LSM, as described before. The logic-containing portion is referred to as the *security server*, and it is by using this component that policy makers can specify the access controls for the system. The enforcement mechanism portion is referred to as the *object manager(s)* whose job is to correctly label and protect access to all the objects in the system.

Flask has two policy-independent data types that are used by the system to label subjects and objects. The *security context* is a variable length string representing the label of the entity while the *security identifier (SID)* is an integer that the security server maps to a security context. In SELinux, a security context contains a user identity, role (only for processes), type, and MLS level (if implemented). A SID is only provided if the combination of identity, role, type and level is valid for the system (*user:role:type*). An example security context is

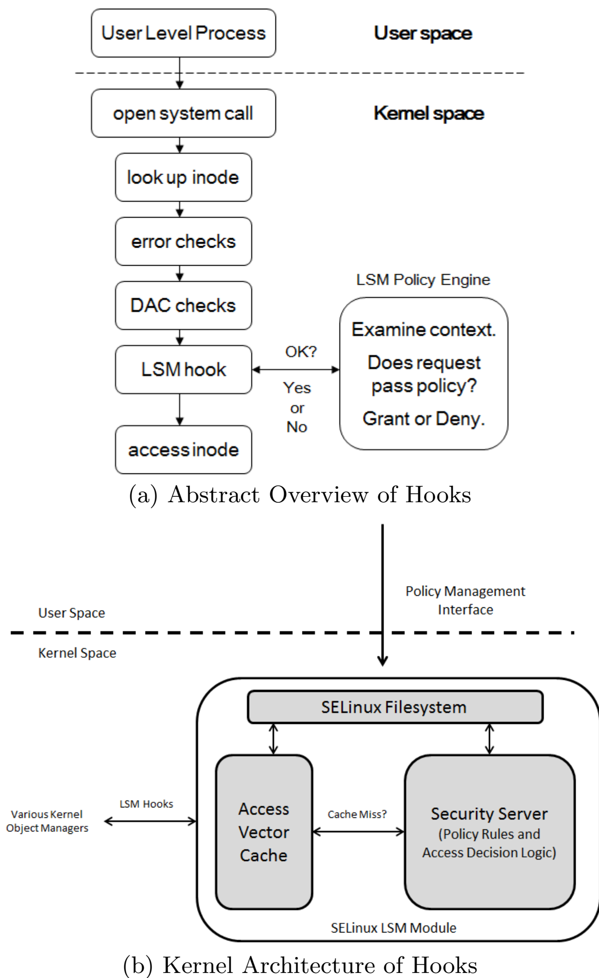
`system_u:system_r:httpd_exec_t`.

This means that the system 'user' has the root role and can run the files of type *httpd_exec_t*. The LSM only has a single void* security field in the kernel data structures for the kernel objects, and since SELinux needs two (security context and SID), it stores the information in a dynamically generated structure for each kernel object. The SELinux policy consits of rules that define types and transitions, and specify what domains are able perform which actions. An example access rule is: "`allow sshd_t sshd_exec_t:file {read execute entrypoint};`". This rule says that the `sshd_t` domain can read, execute, and be entered via a file with the `sshd_exec_type`. The other policy rules follow similar format.

The LSM inserts hooks into important kernel functions that manage objects so that labels can be applied and access permissions can be checked. The hooks implement the conceptual Flask object managers, and whenever an object manager needs a label for an object, the security server is checked to determine what the label should be. For processes in Flask, the label generally depends on the label of the current process and the label of the program's executable. For files in Flask, the label generally depends on the label of the current process, the parent directory's label, and the type of file that is being created. For SELinux, a new process will have a role and domain (type) based on the role and domain of the parent process as well as the type of the program. Additionally, a file will have a type specified based on properties of the process and type of program and can use specified types based on the domain of the process, the type of the parent directory, and the kind of file.

As in DTE, Flask/SELinux performs specialized access checks in addition to and before the standard user-group-world access checks. The Flask object managers, or hooks in LSM terminology, are responsible for performing the checks. Figure 7 shows the LSM hook archictecture. Figure 7(a) shows abstractly where the hooks occur in access checks and Figure 7(b) shows the kernel architecture.

To perform checks, the object managers consult the access vector cache (AVC) which stores access decision computations that the security server has completed for further use by the object managers. The object managers check rights by giving the AVC a pair of labels (for the subject and ob-



(a) Abstract Overview of Hooks

(b) Kernel Architecture of Hooks

Figure 7: SELinux Hooks [20]

ject) and an object class. If the AVC has already calculated a permission based on these attributes, it is returned. If not, then it contacts the security server to obtain the permissions and stores the results for later use. The object class is defined for type of object and each object class has a set of permissions which are specified in a bitmap called *access vector*. Each service has separate permissions defined for each class of objects it accesses. SELinux defines many types of permissions, including *execute*, *transition*, and *entrypoint* which are similar to the domain transitions in DTE (a process can only transition to another SID if it has been appropriately setup with transition and entrypoint permissions). Other permissions involve signal handling, forking, tracing, and obtaining/changing information on the process.

4. COMMON PROBLEMS AND GENERAL BEST PRACTICES

The access control mechanisms discussed in this paper vary from simple to extremely complex. Each system, even if they are implementing the same types of access control (DAC or MAC), have variances in the details of the implementations. With all of these differences it becomes very important for the application developers to fully understand how the access controls on their target system(s) are implemented so that they can be used appropriately and most effectively. However, access controls are improperly used in systems all the time, and improper use can cause serious harm [12].

Common Weakness Enumeration (CWE) is an effort by MITRE Corporation to incorporate all of the research into software weakness by different research groups into a single resource to be used by researchers, analysis tool designers, and application developers alike (`cwe.mitre.org`). The weaknesses are grouped into common types of problems based on the nature of the weakness. For access control, many examples of misuse of ACLs further show the need for developers to understand their operating environment to protect systems from malicious actions. Some of these CWE entries from version 1.12 will be presented.

CWE 266 contains examples of incorrect privilege assignment, where a product assigns a privilege to an actor that allows that actor have unintended ability in the system. One example is from the Apple Mac OS X 19.3.9 system, specifically the authorization services in `securityd`. The security context daemon `securityd` maintains security contexts and is the arbiter for all cryptographic operations and security authorizations. The weakness is that it allows users to grant themselves rights that should be restricted to Administrators. This weakness is currently under review, meaning that it hasn't been accepted as an official weakness. However, it points out a common problem with access controls. In a stand-alone system, this might not be considered a weakness and instead be considered a necessity (i.e., a user on his/her own machine needs to have administrator rights to be able to manage it). However, in a network of machines where the network administrators are charged with restricting the rights of users on the local machines, this is definitely a problem.

CWEs 250 and CWE 271 describe weakness involving executing a process/program with unnecessary privileges (CWE 250) and not dropping or lowering privileges (CWE 250 and CWE 271). This generally means that a program is running with more privileges than are necessary for it to perform its functions (i.e. it violates the principle of least privilege). This violation is a common occurrence as developers sometimes think that they need more privileges than they really do and/or it is easier to just claim many or all privileges rather than exactly what privileges are needed [12]. One example given for CWE 250 is in the `splitvt` program [1], which will split a terminal vertically into two shell windows. It executes another program called `xprop` without dropping the privileges, which allows local users to gain privileges. This entry is also still under review, but a look at the project's page shows that it suffered from many security holes that led to improper privileges being obtained. An example given for CWE 271 is the `ping` program in `iputils` as distributed on Red Hat Linux 6.2 through 7J. The program does not drop privileges after receiving a raw socket which leaves it exposed to bugs that could not occur at lower privileges.

CWE 267 describes a similar issue to CWE 250. In 267, a privilege is defined with unsafe actions meaning that a particular privilege, role, capability, or right can perform actions that the developer did not intend, event though it

was assigned to an entity correctly. One example is a vulnerability in Microsoft Internet Explorer 5.0, 5.01, and 5.5 that allows a remote attacker, via some improperly defined right, to monitor the contents of the Windows clipboard.

CWEs 276–279 all relate to a product's initialization and inheritance of permissions. CWE 276 gives examples of products who set incorrect permissions on their object at installation, allowing it to have more access than it should. CWE 277 describes insecure inherited permissions, where a product insecurely defines permissions which are then inherited to all objects that it creates. CWE 278 shows how products can inherit insecure permissions for objects when performing actions like copying from an archive file. CWE 279 gives examples of products that while executing, set permissions for objects in direct contradiction to the permissions the user had specified for the object.

These CWE entries mentioned are just a subset of all the entries that pertain to permissions, privileges, and access controls, and an even smaller subset of all the weakness defined in CWE. However they point out several issues that application developers should be aware of when writing applications to avoid making similar mistakes. Other sources of information about access control program include books on writing secure code. In his book *Programming Windows Security*, Keith Brown presents the general strategies that operating systems (not just Windows) use to perform access checks as a function of the amount of information available [3]. The three types of information that he describes are:

1. The authorization attributes for the principle requesting access
2. The intentions specified in the request
3. The security settings for the object to be accessed

Then, depending on how much of this information is available to a system, different strategies can be imposed. When only (1) is available, an application will likely use impersonation, where the client's token is passed onto the process that is performing the access. In this scenario, the decision is left up to the underlying system as to whether or not the process (and therefore the user) can access the object. When (1) and (2) are available, the application can perform role-centric checks where the client's token should specify if the action is allowed for the application. If all three are available, then the application can enforce object-centric strategies like those in Windows, where each object has an associated security descriptor and the privileges in the client's token are compared with the DACL on the object. These strategies are not all-encompassing, but they are at least a good starting point when thinking about how the target system performs access controls.

Michael Howard, in his book *Writing Secure Code*, discusses a general method for determine what types of ACLs are needed for an application [12]. He is a strong advocate for determining the least amount of privilege an application needs and using that to create the ACL. In addition, he advocates having an understanding of and being accountable for every ACE in an ACL. He gives four basic steps for determining appropriate ACLs:

- Determine the resources that will be used
- Determine the business-defined access requirements for the resources of the application
- Determine the appropriate access control system technology that meets the needs of the access control requirements
- Convert the requirements into the technology

In addition to these general guidelines, Howard also points out some Windows-specific mistakes that programmers often make. One mistake is not getting the order of the ACEs correct when specifying ACLs in code. When working with the Windows GUI to specify ACL entries, the entries are automatically placed in deny-first order so that deny takes precedence over allow. This is not the case when specifying ACLs through code, so developers must be sure to use the following order: explicit deny, explicit allow, inherit deny from parent, inherit allow from parent, inherit deny from grandparent, inherit allow from grandparent, etc. More importantly, developers should be aware that specifying a NULL DACL will grant all access to all users, and NULL DACLs should *never* be used in applications.

Finally, Matthew Bishop in his book *The Art and Science of Computer Security* poses questions that developers should keep in mind when using ACLs on a system [2]. These are as follows:

- What subjects can modify ACLs? *(Usually, this is the owners of the objects)*
- Do the ACLs apply to users of privilege? *(Usually only in a limited fashion)*
- Are groups or wildcards allowed?
- If two ACEs are contradictory, how is the access resolved?
- What are the default settings if a user is not mentioned specifically in an ACE and how/when will the settings be changed?

5. CONCLUSION

A main line of defense against attackers are the access control mechanisms that operating systems employ. Without a thorough understanding of how a given operating system implements access controls, application developers might inadvertently create vulnerabilities in their applications that allow attackers to obtain elevated rights in the system. By using the information in this paper regarding how operating systems implement access controls, the common mistakes that are made, and the general guidance for using access controls in applications, application developers can make their products more robust and protect user data.

6. ACKNOWLEDGMENTS

I would like to thank Jeff Foster and Mike Hicks for their invaluable help in this research and reviewing the paper.

7. REFERENCES

[1] splitvt project page. Accessed from: http://freshmeat.net/projects/splitvt/ April 25, 2011.

[2] M. A. Bishop. *The Art and Science of Computer Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[3] K. Brown. *Programming Windows Security*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[4] M. Corp. Access control. Accessed from: http://msdn.microsoft.com/en-us/library/aa374860(v=VS.85).aspx April 16, 2011.

[5] M. Corp. A history of windows. Accessed from: http://windows.microsoft.com/en-US/windows/history April 16, 2011.

[6] M. Corp. How security descriptors and access control lists work. Accessed from: http://technet.microsoft.com/en-us/library/cc781716(WS.10).aspx April 16, 2011.

[7] M. Corp. What does internet explorer protected mode do? Accessed from: http://windows.microsoft.com/en-US/windows-vista/What-does-Internet-Explorer-protected-mode-do April 16, 2011.

[8] S. Govindavajhala. Windows access control demystified. Technical report, 2006.

[9] A. Grünbacher. Posix access control lists on linux. 2003.

[10] S. E. Hallyn and P. Kearns. Domain and type enforcement for linux. In *4th Annual Linux Showcase and Conference*, pages 247–250, 2000.

[11] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman. Protection in operating systems. *Commun. ACM*, 19:461–471, August 1976.

[12] M. Howard and D. E. Leblanc. *Writing Secure Code*. Microsoft Press, Redmond, WA, USA, 2nd edition, 2002.

[13] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[14] F. Mayer, K. MacMillan, and D. Caplan. *SELinux by Example: Using Security Enhanced Linux (Prentice Hall Open Source Software Development Series)*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[15] M. Russinovich and D. A. Solomon. *Windows Internals: Including Windows Server 2008 and Windows Vista, Fifth Edition*. Microsoft Press, 5th edition, 2009.

[16] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals, Fourth Edition: Microsoft Windows Server(TM) 2003, Windows XP, and Windows 2000 (Pro-Developer)*. Microsoft Press, Redmond, WA, USA, 2004.

[17] R. Sandhu and P. Samarati. Access control: principles and practice. *Communications Magazine, IEEE*, 32(9):40 –48, Sept. 1994.

[18] S. Smalley, C. Vance, and W. Salamon. Implementing selinux as a linux security module, 2002.

[19] M. M. Swift, P. Brundrett, C. Van Dyke, P. Garg, A. Hopkins, S. Chan, M. Goertzel, and G. Jensenworth. Improving the granularity of access control in windows nt. In *Proceedings of the sixth ACM symposium on Access control models and technologies*, SACMAT '01, pages 87–96, New York, NY, USA, 2001. ACM.

[20] C. Wright, C. Cowan, and J. Morris. Linux security modules: General security support for the linux kernel. In *In Proceedings of the 11th USENIX Security Symposium*, pages 17–31, 2002.