SECURE ACCESS CONTROL IN UNIX

HEMAL NAMDEV RATHOD

(B.E. - Computer Science & Engg.)

A THESIS SUBMITTED

FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2004

Acknowledgments

I would like to express my gratitude to my supervisor Assoc. Prof Roland H. C. Yap for his valuable guidance throughout my research work. I am grateful to him for instilling in me the art of thinking diversely while tackling issues during research.

I would also like to thank my friends Vijay Kothari, Arun Shenoy and Roshni Mohapatra for supporting me morally and academically during the course of my study.

To my beloved parents: For whatever I am today, its because of you and I will remain grateful to you all my life.

- Hemal

Contents

List of Tables

List of Figures

Summary

Computer Security has been a chronic problem since a very long time. Several approaches have been employed but are usually flawed because they rely on existing security mechanism of the mainstream operating systems. Security needs to be addressed at the operating system level. Unix, in its various forms uses access control lists(ACLs) to achieve access control which is not sufficiently powerful to achieve security at the higher level. Unix also does not conform with the principle of least privilege. Various other mechanisms and models have been developed recently for access control in Unix based systems. In this research, we identify the major problems with the current Unix model, outline the various research techniques which are employed to address them and present our solution which achieves more flexibility and a more fine grained model.

Our approach is to use capabilities for achieving access control. Capabilities have been used several times in the past for achieving access control. We have altered the traditional capability structure to create environments to confine processes in a domain. These environments, called CBoxes can be used to provide confinement since processes can only access objects whose capabilities are available in that CBox. Capabilities can also be added and dropped from the CBoxes dynamically thereby achieving dynamic access control. We have defined several other types of capabilities like one-time capabilities and time-stamped capabilities to achieve more flexibility. Fine grained access permissions are defined for each type of object in the system so that capabilities for only those fine grained permissions can be issued and nothing more. Not only are the objects accessed using capabilities, we also define resource capabilities which are used for limited usages of resources. We also address the issue of the all powerful 'root' in the Unix system and propose methods of delegation of power using capabilities. We have implemented our model in a Linux system and carried out performance evaluation. We

show how capabilities can be used to implement higher level abstraction such as RBAC.

Chapter 1

Introduction

Over the past several years, the internet environment has changed drastically. This network which was once populated almost exclusively by cooperating researchers, who shared trusted software and data, is now inhabited by a much larger and more diverse group that includes pranksters, crackers and business competitors. Since the software and data exchanged on the internet is very often unauthenticated, it could easily have been created by an adversary [Gol96].

Computer security has been a chronic problem since a very long time. Microsoft's windows operating system have had several security attacks since it was developed. The infamous "Melissa" virus infected thousands of computers with alarming speed, infected Microsoft Word documents on the user's hard drive, and mailed them out through Outlook to the several recipients. The "I Love You" virus infected millions of computers virtually overnight, using a method similar to the Melissa virus. The virus also sent passwords and user names stored on infected computers back to the virus's author. The first virus developed was actually a Unix one - called the "Morris" worm that invaded ARPANET computers and disabled roughly 6,000 computers on the network.

Surprisingly, the computer security problem might not have a standard solution to

it. Anti-virus software and Firewalls have limited applicability to the problem. The increased awareness of the need for security has resulted in an increase of efforts to add enhanced security to computing environments. However, most of the efforts to do so suffer from the flawed assumption that security can adequately be provided without certain security features in the operating system. In reality, operating system security mechanisms play a critical role in supporting security at higher levels. The computer industry has not accepted the critical role of the operating system to security, as evidenced by the inadequacies of the basic protection mechanisms provided by current mainstream operating systems [Los98].

Access control forms the core of system security in an operating system. Access control is any mechanism by which a system grants or revokes the right to access some data, or perform some action. Normally, a user must first Login to a system, using some Authentication system. Typically, the Access Control mechanism controls what operations the user may or may not make by comparing the User ID to an Access Control database. Access control mechanisms (User/Group IDs and ACLs) in mainstream operating systems like Unix and Windows are inadequate to provide high level of security. The Unix kernel only provides discretionary access controls and lacks any direct support for enhanced access control mechanisms. In Windows, protection is largely based on access control lists.

Variations of Lampson's access matrix [Lam73] in the form of access control lists (ACLs) have been used as the security mechanism for basic access to objects. Unix in its various versions also does the same either at the granularity of user/group/world or some Unix have ACL. However the basic ACL mechanism is not sufficiently powerful to obtain a high level of security. Furthermore, there are security problems which arise from exploiting the fact that privilege users such as root have too much access. The root cause can often be traced to not using the *principle of least privilege* [Sch75]. Applying least

privilege however is problematic since that requires a more powerful security mechanism in place and it cannot be achieved using ACLs. There have been a number of basic approaches to the problem of too much privilege. One way is to use mandatory access control mechanisms (MAC). Some examples in Linux are SELinux [Los01], LIDS [Xie01] and RSBAC [Ott01]. Another approach is to confine an untrusted process by using sandboxing techniques. Again there are numerous systems, such as Janus [Ian96] and Systrace [Pro02]. Capabilities [Den65] were invented as a fine grained mechanism for protecting objects. While it has been used in the past in various secure operating systems [Dat89, Coh75, Tan86], it is fair to say that existing mainstream operating systems do not employ capabilities as their security mechanism. In short, it seems to have fallen out of use. There is recent work on capability based operating systems, such as KeyKOS [Har85] and EROS [Sha99], however these operating systems are designed from the ground up to be based on capabilities and are thus not compatible with existing operating systems. This is a major issue when it comes to usage, compatibility, porting and availability of applications. One reason for the lack of interest for a long time in capabilities as a basic security mechanism may be due to several myths which have been associated with capabilities. Miller et al. [Mil03] describes the three important misconceptions:

- The equivalence myth: ACL and capability systems are formally equivalent.
- The confinement myth: Capability systems cannot enforce confinement
- The irrevocability myth: Capability systems do not support revocation

It is more correct to say that these myths depend very much on the precise properties of the capability system and how it is used (in the operating system or otherwise). It is useful to note also that other solutions such as MAC and sandboxing may not address these issues well either. Revocation in particular may not make sense in either MAC

or sandboxing. MAC however does not directly address confinement while sandboxing does. We propose a new capability based system which we have designed to be essentially compatible with Unix. Compatibility means that it is possible to setup the environment and Unix to take advantage of the extra security introduced with capabilities and yet allow existing Unix applications to remain unchanged (binary compatibility) and also keep the feel of Unix unchanged. Applications could be written of course to get increased security using these mechanisms but it is possible to achieve that even with existing binaries. Unlike the myth, our system can be used to provide confinement. However it goes beyond confinement since capabilities can be used to produce either a highly confined environment which is similar to a sandbox or a more loose environment where collections of processes can share access in a cooperative fashion. For example, based on their own computation several processes could exchange capabilities or even trade resources such as CPU time, disk quota, etc. Unlike systems like EROS, we achieve more compatibility with regular Unix behavior and yet provide both fine grained access control as well as confinement.

In the next chapter we discuss the background and related work. Design goals and approach is discussed in chapter 3. A detailed design of the capability based mechanism is outlined in chapter 4. Implementation details and performance evaluation is discussed in chapter 5. In chapter 6 we show how capability based mechanism can also be used for Role Based Access Control. Chapter 7 outlines the conclusion and future work.

Chapter 2

Background and Related Work

This chapter is divided into two sections. The first section discusses the various motivation points and problems existing in current mainstream operating systems. The later sections discusses the related work to indicate various approaches taken to provide security.

2.1 Motivation

2.1.1 Inadequate Operating System Support

Operating systems like Unix were not designed to protect against current security issues arising due to networked environments. Hence, it has various loopholes leading to security being compromised. We elaborate the problems with the current Unix security model.

Principle of least privilege

The lack of flexibility in modern operating systems is one of the main reasons security is compromised. In addition, Unix allows the granting of temporary privileges, namely setuid(2) and setgid(2). Special care must be taken any time these primitives are used

since it is well known that it is difficult to write secure programs. Access decisions are made on the basis of the effective userid of the process. An attacker can utilize a flaw in the program and could use inputs much larger than the defined size leading to buffer overflow.

Coarse Granularity

There is the small granularity of discretionary access rights only dividing between read, write and execute rights for file owner, file group members and all others. The fact that access control relies on a file owner's discretion already leads to various problems, like the level of trust that has to be put in a user, the vulnerability from malware working on behalf of a user, etc.

All Powerful Root

The worst problem however is the system administrator account 'root'. Numerous system tasks are only allowed to be done by this user, even various network services have to be started or, worse run as root. The root account has full access to every object in the system. This is one of the main reasons many Unix family systems have been compromised locally or by remote access. For example, when an attacker is running a program owned by root, it can utilize any loopholes in the program which leads to execution of a shell. Since the root can access all objects in the system, the attacker can also do anything if he can obtain a root shell.

These mechanisms are inadequate to handle the complex security needs of today's applications. Hence, extra security is needed for the user to run these applications safely.

2.1.2 The Preexisting Reference Monitor

The traditional Operating System's monolithic reference monitor cannot protect against attacks on applications directly. It could at most, prevent a penetration from spreading to new accounts once the user's account has been compromised, but by then the damage is already been done. In practice, against a motivated attacker most operating systems fail to prevent the spread of penetration; once one account has been subverted, the whole system typically falls in rapid succession.

2.1.3 The Conventional Network Firewall

Packet filters cannot distinguish between different types of HTTP traffic, let alone analyze the data for security threat. A proxy could, but it would be hard-pressed to understand all possible file formats, interpret the often-complex application languages, and squelch all dangerous data. This would make for a very complex and thus untrustworthy proxy. Firewalls only provide some protection at network packet level and no protection against malicious insiders. Typically, insiders can easily leak information through the firewall.

2.2 Current Approaches

2.2.1 Mandatory Access Control

The philosophy underlying discretionary access control (DAC) is that the owner or administrator of the information has the knowledge, skill, and ability to limit access appropriately, to control who can see or work with the information. A person running a working group could provide working group members with access to shared files but deny access to anyone else. Alternatively, and more frequently, global access is provided by default.

In MAC, information is categorized according to sensitivity rather than subject matter. Data in the same general subject matter area can exist in files with different sensitivity concerns. People and processes within a MAC-managed environment are adjudicated as to what kinds of sensitivity levels they are allowed access to. The MAC processes enforce these access limitations. For example, information classified secret is accessible only by people or processes adjudicated for access to secret or more highly secret information. People or processes not adjudicated for secret-level information would be barred from accessing it.

Various systems have used MAC as their design principle. In this section we describe approaches of a few systems like SELINUX [Los01] and RSBAC [Ott01].

DTE

A DTE [Lee96] system associates a domain with each running process and a type with each object (e.g., file, packet). As a DTE UNIX system runs, a kernel-level DTE subsystem compares a process's domain with the type of any file or the domain of any process it attempts to access. The DTE subsystem denies the attempt if the requesting process's domain does not include a right to the requested access mode for that type.

SELINUX

Secure-Enhanced Linux [Los01], or SELinux for short, is an application of the Flask architecture in the Linux operating system. MAC has been integrated into the major subsystems of the Linux kernel, including fine-grained controls for operations on processes, files and sockets. The security policy decision logic has been encapsulated into a new kernel component called the Security Server (SS) which makes labeling, access and polyinstantiation decisions in response to policy-independent requests that have been placed throughout the kernel. This architecture enables the kernel to enforce

policy decisions without needing access to the details of the policy. SELinux is designed around two key concepts: fine-grained mandatory access controls and a separation between enforcement mechanisms and policies.

SELinux ensures that every system call is authorized as a function of:

- The running program
- The user running the program and
- The context that invoked the program

Similar to DTE, the security policy configuration in SELinux uses a high-level language for specification. Type enforcement can be combined with Role-Based Access Control (RBAC), wherein each process has an associated role. System processes run in the system_r role, while user may have roles such as sysadm_r and user_r. A role is only allowed to enter a specified set of domains. Domain transitions may occur, for example, to a more restricted domain when the user runs an application that should be run with a subset of the user's privileges.

A policy could be designed so that a program could only read and not write to the filesystem while at the same time have the ability to write to log files, but when running the same program as a different user have the program be able to write to the filesystem.

Security Server: The flexibility of the Flask Architecture allows the security server to be modified, or even replaced, to alter the supported security model to meet additional requirements. The content and format of labels used in the system depend on the particular security model implemented by the security server. Security decisions within the security server are based on security contexts which represent security labels. A security context is a policy independent data type that can be handled by different parts of the system but should only be interpreted by the security server. It contains all

of the security attributes associated with a particular labeled object which are relevant to the policy decision login.

Security contexts are usually not bound directly to objects. A second policy-independent data type called a security identifier (SID) is bound to each object that requires a label. SIDs are nonglobal and nonpersistent opaque objects that are mapped to security contexts. This mapping is created at run time maintained by the security server. When an object is created, the security server decides which SID to use as a label. SIDs associated with objects are passed into the security server and used as the basis for security decisions.

The mandatory access controls of SELinux are implemented as permission checks that have been inserted as control points throughout the Linux kernel. Approximately 140 fine-grained permissions, grouped into 28 object classes, have been defined to allow the control of nearly every system operation. Permission checks are made between a source SID and a target SID for a particular permission in some object class. Usually, but not always, these are the SIDs associated with a calling process and some object, like a file, that is being accessed. To respond to permission checks, the security server's policy logic uses the security relevant attributes contained in the security contexts associated with the source and target SIDs to determine if permission can be granted.

RSBAC

RSBAC [Ott01] is a flexible, powerful and fast open source access control framework for current Linux kernels. The RSBAC framework is based on the Generalized Framework for Access Control (GFAC) by Abrams and LaPadula. All security relevant system calls are extended by security enforcement code. This code calls the central decision component, which in turn calls all active decision modules and generates a combined decision. This decision is then enforced by the system call extensions.

Decisions are based on the type of access (request type), the access target and on the values of attributes attached to the subject calling and to the target to be accessed. Additional independent attributes can be used by individual modules, e.g. the privacy module (PM). All attributes are stored in fully protected directories, one on each mounted device. Thus changes to attributes require special system calls provided.

As all types of access decisions are based on general decision requests, many different security policies can be implemented as a decision module. Apart from the built-in models, the optional Module Registration (REG) allows for registration of additional, individual decision modules at runtime.

The RSBAC framework gives detailed access control information, and you can implement almost any access control model in it, e.g. as a runtime registered kernel module. Also, there is a powerful logging system which makes intrusion attempts easily detectable.

2.2.2 Role Based Access Control

The Role-Based Access Control (RBAC) model provides a flexible method for managing access control in complex systems. It has been widely used for access control in many environments including organizations, databases and operating systems. Implementing and administrating the RBAC model is an important issue since the model can contain large number of users, roles, permissions and constraints.

RBAC96

Four different models RBAC0, RBAC1, RBAC2, RBAC3 were described in the RBAC96 paper by Sandhu et al. [San96]. There are three sets of entities called users, roles and permissions. A user can be a human being or agents like robots and computers. A role is a job function or job title within an organization. A permission is an approval of a

particular mode of access to one or more objects in the system. Users are assigned to roles, permissions are assigned to roles and user acquire permissions by being member of roles. The user-role and permission-role assignment can be many-to-many. A sessions is defined as a mapping of one user to many roles. A user can establish a session and activate a subset of roles assigned to the user. During this session the user can utilize the union of the permissions assigned to the roles which are active. Each session is associated with a single user and it remains constant for the life of a session.

Hierarchical RBAC

Hierarchical RBAC organizes roles into a hierarchy. The hierarchy defines a partial ordering defining a seniority relation between roles, whereby senior roles acquire permissions of their juniors. The NIST model [Dav01] recognizes two types, General Hierarchy and Restricted Hierarchy. In general hierarchical RBAC, there is support for an arbitrary partial order to serve as the role hierarchy. In restricted hierarchy, hierarchies are restricted to structures like trees.

Constraints in RBAC

Constraints are introduced in the RBAC model in the form of separation of duties(SOD). Three different types of SOD exist in the form of Static SOD, Dynamic SOD and Historical SOD. Static SOD constrains the assignment of users and permissions to roles. Dynamic SOD constrains the activation of roles and invocation at run time. Historical SOD constrains the invocation of permissions over the course of time.

Administration

Managing RBAC includes managing the roles, their interrelationships, assigning users and permissions to roles and various other complex administrative tasks. RBAC96

[San96] proposed using RBAC itself to manage RBAC. This is called ARBAC where A denotes administrative. Other methods proposed are ARBAC97 and its improved model ARBAC02 by Oh and Sandhu [Oh02]. These models are used mainly for organizing role hierarchies.

2.2.3 Sandboxing

A widely used technique for securing computer systems is to execute programs inside protection domains that enforce established security policies. These containers, often referred to as sandboxes, come in a variety of forms. Although current sandboxing techniques have individual strengths, they also have limitations that reduce the scope of their applicability. This section describes in brief various research done on this technique.

System Call Interposition

Janus [Ian96] is a secure environment for untrusted helper applications built by taking advantage of the Solaris process tracing facility. Janus intercepts and filters dangerous system calls via the Solaris process tracing facility. An application can do anything it likes that does not involve a system call. A configurable policy decides whether system calls like *open* and *rename* can be allowed or denied.

Consh [Ale99] incorporates a modified version of Janus as one of its modules. Consh virtualizes all the resources visible to untrusted applications which allows protected resources such as files and directories or networking to be replaced with safe alternative local or remote ones transparently. It works by intercepting the system calls issued by the applications and modifying their behavior. To provide protection, Consh aborts system calls that attempt to access forbidden resources. To provide access to remote resources, Consh handles or modifies the behavior of system calls that access such resources.

Systrace [Pro02] enforces system call policies for applications by constraining the

application's access to the system. The policy is generated interactively. Operations not covered by the policy raises an alarm, allowing a user to refine the currently configured policy. The Systrace project is similar to Janus, but Systrace has a GUI monitor program to manage policies and is able to modify policies at run time.

Information critical to sound security policy decisions is not available at *system call interposition*. At the system call interface, userspace data, such as a path name,has yet to be translated to the kernel object it represents, such as an inode. Thus, system call interposition is both inefficient and prone to time-of-check-to-time-of-use (TOCTTOU) races [Bib96].

Monitor Based Sandboxes

Monitor based sandboxing techniques uses a process to monitor a particular sandbox. [Pet02] illustrates one such mechanism which uses monitors to control the sandbox. The sandboxing mechanism in this system is implemented as a system call API that serves as a general-purpose framework for confining untrusted programs. It supports nested monitor (or nested sandbox) and uses monitors to manage ACLs. The monitors are used to manage all policies. For each syscall from the confined process, the kernel will asks the monitor to decide allow/deny.

Other Sandboxing Techniques

MAPbox [Ach99] focuses on classifying application behaviors in order to implement an application-class-specific sandboxing mechanism. The key idea is to group application behaviors into classes based on their expected functionality and the resources required to achieve that functionality. Applications are classified into Behavior classes, such as reader, compiler, editor and shell.

BSD Jail - *Jails* [BsdJa] are typically set up using one of two philosophies: either to

constrain a specific application (possibly running with privilege), or to create a "virtual system image" running a variety of daemons and services. Jail can be used to restrict a process to access only a specific directory tree. A fairly complete filesystem install on the directory tree is required. This feature is similar to *chroot*(). In addition, we can use jail to restrict a process to access (e.g. bind, listen) a specific network interface.

FMAC [Pre01] describes a portable system that tracks the file requests made by applications creating an access log. The same system can then use the access log as a template to regulate file access requests made by sandboxed applications.

2.3 Existing Capability Mechanisms

Capabilities [Den65] were invented as a fine grained mechanism for protecting objects. A capability list can be realized by vertically compressing the Access Control Matrix as defined by Lampson [Lam73]. A capability can be defined as a ticket which has the name or the identity of an object and the access rights. The holder of the capability can access the object with the access right on the capability upon authentication of the capability.

Several systems using the capability concept have been marketed (IBM System 38, CAP, i432, Plessey S250) [Lev84]. These capability based systems have been used only for certain types of systems and require support of special hardware. In traditional operating systems, capabilities were created and managed only by the kernel processes and stored in the privileged system space. Distributed capability based systems like Amoeba [Tan86] used capabilities in the user space. Amoeba uses cryptographic techniques for ensuring the integrity of capability in user space. The commonly used method is to append to the capability a *check* field which is usually the output of a one-way hash function applied to all data on the capability, and only the object manager has the knowledge to recompute this field and verify its genuineness.

[Kao96] proposes an *extended capability* architecture which extends the traditional capability scheme to enforce as many dynamic access control policies as possible while keeping the extra overhead minimum. It embeds the *owner* into the capability similar to the mechanism proposed by Gong [Gon89] so that malicious users cannot use them illegally. It also embeds an *expire* field indicating the time at which the capability expires. Access control information like policy number and policy dependent information is also added to the capability. A *check* field is used for protecting the capability from forgery.

The basic idea in *Split* capabilities [Kar03] is to divide the capability into two parts, a handle to the resource being accessed and a handle to a separate resource representing the access rights being requested. Using split capabilities with visibility controls allows the simple specification of complex security policies.

KeyKos [Har85] and EROS [Sha99] are capability based operating systems and are created from the ground up. EROS is a pure capability system. Authority in the system is conveyed exclusively by secure capabilities, down to the granularity of individual pages. Some features of EROS are as follows:-

Orthogonal Global Persistence - All user state, including both data and running programs, are transparently saved on a periodic basis. In the event of system failure processes are resumed as of the last checkpoint. No special action or programming on the part of the application is required.

Kernel Threads - The EROS kernel itself is implemented using multiple kernel-mode threads. This improves the performance of EROS drivers, makes them simpler to code, and greatly simplifies the design of the kernel. In addition, it enables selected kernel functionality to be preempted by higher priority user activities.

Security - Because EROS processes are persistent, processes can hold authorities in their own right rather than inheriting them from the user. This enables a rich variety of

options for security and access control that are impossible in systems lacking persistent processes.

Chapter 3

Design Goals and Approach

3.1 Fine Grained Security

File protection in a UNIX system is achieved by using Read, Write and eXecute permissions at three different levels, the file owner, other users in the owners group and the rest of the world. The UNIX protection mechanism, therefore, is coarse grained and cannot affect objects whose access has been obtained, eg. open files.

3.2 Features of both DAC and MAC

The philosophy underlying discretionary access control (DAC) is that the owner or administrator of the information has the knowledge, skill, and ability to limit access appropriately, to control who can see or work with the information. In mandatory access control (MAC) and role-based access control (RBAC), management of access is much more structured. Both assume a set of formal rules about who can have access to what kind of information and what can be done with that information. Systems which use MAC rely on administrators to define security. However, the administrators may not always be correct. DAC provides user programmability which can be convenient but has the disadvantage that system administrator policies may not be enforced. One

goal is to provide a combination of both MAC and DAC features.

3.3 Dynamic Support for Policies

One needs a mechanism which supports dynamic changes to the policies not only once but several times for the system while its running. One also needs a mechanism to revoke permissions that are implicitly retained in the state of the system, e.g. open file descriptions, memory-mapped files, established TCP connections, and even operations currently in progress that have already checked permissions.

3.4 Decentralisation

Traditionally, the root account has full access to every object in the system. We want a system where the power can be decentralized to users and root has limited privilege. Hence, even when a root account is hacked into, not much of damage can be done with limited root permissions except to those objects accessible with a root account.

3.5 Flexibility

Most security policies are designed to work towards providing security by restricting the access to a specified set of objects. They are simple mechanisms to use but they often lack the flexibility. Therefore the problem lies in understanding where the compromise is.

3.6 Sandboxing

A sandbox provides a safe environment for programs to execute in. A sandbox is an environment in which the actions of a process are restricted according to a security policy. Untrusted applications and programs which are possibly infected with virus can

be mitigated by means of sandboxing. The sandboxing of application would provide fine grained confinement for multiple applications.

3.7 Why Capabilities?

We have modified the traditional capability structure and this allows us to attain the goals that we have mentioned in the previous section. We discuss how they can be achieved using capabilities.

3.7.1 Confinement

Using capabilities for access control provides confinement. Capabilities can be used to confine a particular process by restricting it in an environment in which access to objects is limited to the capabilities available for those objects. For example, if a process running a compiler is attached to an environment containing read capabilities for compiler program files, only those files can be accessed and nothing more. If more confinement is required, capabilities can be dropped to create a tighter environment. Only objects whose capabilities are available can be accessed and nothing more. These capabilities are neither forgeable nor can they can be exploited outside the desired process. They cannot be propagated illegally as will be explained in detail in the sections describing capabilities. Hence these capabilities can be used to provide confinement and conforms with the principle of least privileges.

3.7.2 Fine Grained Security

Most of the mechanisms available do not provide fine grained security. Access to each object in a system needs to be fine grained and using capabilities makes it easier for a fine grained security model. These capabilities are then used in environments for accessing objects with only those fine grained permissions which are present in the capabilities

and nothing else.

3.7.3 Decentralisation

Since access to each object requires a capability, root may no longer possess all of them, thus root is restricted. Capabilities for objects and permissions can be appropriately propagated to various users and the root might just possess capabilities for certain objects which it requires. This way root cannot access all the objects in the system and can access only those which required.

3.7.4 Dynamic Control

Capabilities can be added and dropped from environment to provide dynamic control of access rights. Providing access to objects and revoking the rights is just a matter of adding and dropping capabilities from environments. Dynamic control of capabilities provides dynamic access control.

Various other types of capabilities are defined to achieve other features in the capability based mechanism. One-time capabilities can be used only once and becomes invalid after that. There could also be time stamped on each capability so that these capabilities expire after certain time. There might be resource constraints requirements in a program or access to certain resources needs to be limited. Resource capabilities are used to achieve that.

Chapter 4

CBox: Capability Based Sandboxing

4.1 Capabilities

A capability as defined by Dennis [Den65] is an unforgeable $(object, type, authority)$ triple. The *object* in capability refers to object for which access is being controlled, *type* field refers to the type of the object(file, directory etc.) and *authority* refers to the mode in which the object can be accessed. To access any object in a capability based system, a user or a process must possess a capability for it. Possessing a capability is a sufficient and necessary proof for performing the operations. This section illustrates the capability based system which we have designed and explains how it can be used to provide secure access control and confinement. The traditional capability structure is modified so that it can neither be forged nor can it be propagated illegally. Chapter 2 discusses the traditional capability based systems being implemented. We use these capabilities to create dynamically created sandboxes [Ian96],[Pet02],[Ale99],[BsdJa]. These sandboxes, explained in later sections, are called *CBoxes* since they can either be used it as a tight environment for confinement or a loose environment where capabilities can be added

dynamically. Various types of capabilities are used depending on the user and owner of capabilities. Capabilities are also categorized depending on the type of objects in the system. In the next section we discuss the types and structures of various types of capabilities available.

4.2 Capabilities : Structure and Types

The capabilities are categorized into Master and Derived depending on its structure and where it is being used. Structure of each capability is defined in the next section. We also define other types of capabilities which we have designed to provide other features apart from access control.

4.2.1 Master Capabilities

A *Master* capability is like a master key to any lock. The structure of a master capability is as follows:

ObjID	Rights=ALL	Type

The ObjID field contains the identity of the object and is a randomly generated integer. The Rights field is set to 'ALL' since it is the master capability. The Type field is set to the type of object. The master capability is attached to the object. The owner of the object owns the capability and can use this capability to create derived capabilities.

4.2.2 Derived Capabilities

A *Derived* capability is created from the master and is created specifically for a particular domain or environment (called a CBox in our system). These protected domains (CBoxes) are created specifically for a process to use the derived capabilities attached to

the CBox for accessing the objects and resources. The structure of a derived capability is as follows

ObjID	Rights	Stamp	Lease	Properties

The $ObjID$ field is the same as in the master capability and the $Rights$ field can be restricted to a particular right from the set of rights defined for that type of object. The $Stamp$ field is used to stamp the process id (pid) and userID onto the capability to identify which process or user can use it. The set for the stamp field could be defined as follows:

$Stamp = \{\langle UserID, pid\rangle, \langle UserID, *\rangle, \langle *, pid\rangle, \langle *, *\rangle\}$

The first value has stamped the userID and the pid on the capability. This means that only that particular user and process can utilize the capability for access. If the capability is leaked to other users, it is invalid. The second value in the set implies that any process executing on behalf of the user can use the capability. The third value is not generally used since the same pid cannot be used for processes executing for different users. The final value signifies that any user or process can use the capability.

The properties field contains bits which signify whether the capability can be further derived or not. If the 'derived' bit is set to 1 then the capability can be further derived. Note that instead of stamping the user ID alone in the capability as described by Gong, [Gon89] and in [Kao96], we stamp the pid as well so that only the particular process can make use of it. Apart from these fields, the *Lease* field is used to make the capability invalid after the time specified by lease expires. Leasing is useful for time critical capabilities. Leasing was mainly used for revocation in [Kao96].

4.2.3 Encrypted Capabilities

Derived capabilities are encrypted with Advanced Encryption Standard (AES) algorithm to generate *Encrypted* capabilities. The structure of an encrypted capability is as follows:-

ObjID	Rights	Code

The code field in the encrypted capability is obtained by encrypting all the fields in the derived capability using AES. A unique key is used to encrypt the capability.

Capabilities need not be encrypted since we only use references to use the master and derived capabilities in the kernel space. Encrypted capabilities were traditionally used in distributed systems [Tan86] where it needs to be used in the user space to avoid forgery. Encrypted capabilities are used for a different purpose in our system. The derived capabilities are only encrypted when they need to be stored in the filesystem for persistance.

4.2.4 Resource Capabilities

Resource capabilities are capabilities used to control access over resources like CPU time, disk quota etc. The structure of a resource capability is as shown.

ResourceID	Limit	CID	Lease	Properties

A unique ID is given to each resource which is a randomly generated integer similar to ObjID. 'Limit' refers to the amount of resource which can be used by the CBox to which the capability is attached to. This is specific to the underlying object, eg. space, time, number of pages etc. The rest of the fields are similar to the derived capability fields.

4.2.5 Other types of Capabilities

Negative Capabilities

Negative capabilities are specially designed capabilities to realize negative permissions. As the name suggests, possessing such a capability for an object implies that the object cannot be accessed even if a normal capability is available. Negative capabilities (either a master or derived) is used to cancel out the positive capabilities (master or derived capabilities). The structure of negative capability is

CapID	Stamp	Negative	Properties

The CapID is the capability ID of the capability for which this negative capability is created. The stamp field is required to attach it to a particular process or a user so that other cannot use them. Negative field implies that it is a negative capability. The properties field is used to check whether the capability can be further derived. This capability can also be encrypted when it needs to be stored in the filesystem. The structure of an encrypted negative capability is similar to the structure described in the encrypted capability section.

One-time Capabilities

One-time capabilities, as the name suggests, can only be used once. This means that once the capability is used to access the object that it represents, it becomes invalid and using it further would give an error. Several ways of implementing a one-time capability are possible. One way is to use the a separate bit in the properties section of the capability called 'One-time'. When this bit is set, the capability is invalid and cannot be used. Other way of implementing a one-time capability is to revoke the capability after it has been used once. Revocation is discussed in the next section. [Kao96] also mentions one-time capabilities but they are used in user space and hence

are implemented in a different way as compared to our scheme.

4.3 Operations on Capabilities

This section describes various operations that can be performed on the capabilities in this system.

4.3.1 Creation

Master capabilities for an object is created when the object is created. Once the object is created, this master capability is attached to the data structure of the object. A random number is generated for creating the ObjID and the *Type* field is set to the type of the object. If the object already exists, master capabilities are created when the system requests for it. The ObjID could also be the inode number in case the object has an inode.

Creating derived capabilities is done when it needs to be attached to a CBox. The ObjID of the master capability is used to create the derived capability. Appropriate access permission is attached to the derived capability. The pid of process is then stamped onto the capability for which it is meant for and the properties field set accordingly. If the capability can be further derived, the 'Derivable' bit is set to 1. Once the derived capability is created, it is attached to the CBox.

Encrypted capabilities are created by encrypting the derived capabilities using the key for the particular CBox. As described earlier, capabilities needs to be encrypted only when it needs to be stored in the filesystem and not otherwise.

4.3.2 Revocation

Revocation is a process of taking back the rights which were originally granted. Revocation of capabilities is required when any process to whom the capability was granted,

should no longer be able to use it. Revocation addresses a major shortcoming of Unix that it is not possible to revoke access to most objects.

Revocation of a capability from a particular CBox can be done by just dropping the capability from the CBox. Details regarding CBoxes is explained in the coming sections. When a particular process is destroyed, capabilities which are stamped for that process needs to be revoked since they are invalid and of no use. This process of revocation of capabilities for destroyed process is analogous to Garbage Collection.

4.3.3 Propagation

Capabilities attached to CBoxes cannot be propagated illegally to other CBoxes since the capability is valid only for its corresponding CBoxes. This is because the pid or the user id stamped on the capability for which it is actually meant for, hence if it is give to some other CBox, it wont be valid. Propagation can be done among CBoxes if they have the permission to do so. A capability attached to a CBox can be derived further and propagated to other CBoxes. For this, a special API function is available which can be called in order to further derive and propagate. A new capability is then created with the CID of the receiving CBox and other appropriate fields. Depending on the properties field, the capability can be further derived and propagated to other CBoxes.

4.4 Introduction to CBoxes

In our capability based mechanism, capabilities are implicitly used with Unix processes and system calls. Derived capabilities are packaged into CBoxes. A CBox is a composite object which can hold derived capabilities and can be attached to a process running in the system. Environment can be likened to a sandbox created using capabilities. We create our environments using capabilities and call it a CBox. CBoxes are created when a process needs to be restricted. Either a process can be attached to existing CBoxes

or a new CBox can be created for that particular process.

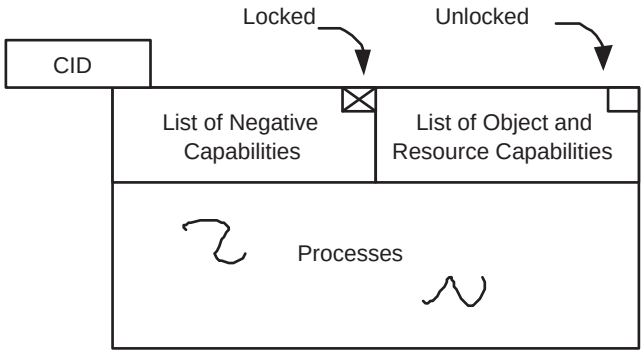


Figure 4.1: Basic Structure of a CBox

A different terminology (CBox) is used since the environment that we create using capabilities can also be a loose environment unlike a tighter environment so it is different from a sandbox. Each CBox has a separate sub section for positive capabilities $perm^+$ (derived and resource) and negative capabilities $perm^-$. Positive capabilities are derived capabilities for objects and resources and are used for accessing them. Capabilities can be added or dropped from the CBoxes depending on whether the CBox is locked or not. If its locked, capabilities can only be dropped to restrict the CBox further and if its in the unlocked state, capabilities can be both added and dropped to the CBox.

Figure 4.1 shows the basic structure of the CBox. As shown, the CBox has a unique ID called the CID. This is a randomly generated integer. This is used as a handle for the CBox created. The CBox has two sub-sections. The Negative capabilities are stored in the negative section. This section is locked, meaning that the negative capabilities cannot be dropped. The other section contains object and resource capabilities. They are derived from the master and can be attached to the CBox. Processes are contained in these boxes and can access objects using capabilities attached to their CBoxes. The notation used for derived capabilities of objects with limited permissions are represented as $deriv(objects.perm)$ and derived capabilities of negative permissions are represented

by $deriv(Negative)$. Stamping of these capabilities with a pid or a UserID is written as $deriv(objects.perm)^{Stamp}$ and similarly for negative and resource capabilities. Hence the CBox containing these capabilities could be represented as

$$CBox(P1) = \{\langle deriv(objects.perm)^{Stamp}\rangle, \langle deriv(resources)^{Stamp}\rangle, \langle deriv(Negative)^{Stamp}\rangle\}$$

where P1 is the process which is attached to CBox.

This CBox itself can be treated as a special type of capability called a *container* capability. This means that this atomic CBox structure could be used to package a set of capabilities and then be attached to a bigger CBox. For example, if we consider two CBoxes, say CBox1 and CBox2, CBox2 could contain a set of positive and negative capabilities and then be attached to CBox1 which has its own positive and negative capabilities. Hence the CBox can now be represented as

$CBox = \{\langle perm^{+}\rangle, \langle perm^{-}\rangle, \langle Caplist\rangle\}$

where $perm^{+}$ is a list of positive capabilities, $perm^{-}$ is a list of negative capabilities and $Caplist$ is a list of CBox container capabilities. All negative capabilities are global and are used to cancel out any positive capabilities in any containers including the container capability.

4.5 Operations on CBoxes

4.5.1 Creation

A CBox is created when a program needs to be executed. Execution of a program involves accessing objects like files, directories etc. and resources. When a user wants to execute an untrusted program, he can choose to initially create an empty CBox. The user can then request for addition of capabilities to the CBox. Depending on which

program is being executed capabilities are derived from the master capabilities attached to the objects and are added to the CBox.

An executable file can have capabilities associated with it. When an exec is called, derived capabilities are created from the master capabilities associated with executable. These derived capabilities are for the executable files which the process needs to access. The *permissions* field of the derived capabilities is appropriately set for successful execution of the program. The capabilities are then stamped with the *pid* or the $userID$ depending on the process and the user executing it. The *properties* field is also set depending on whether the capability can be further derived. If constraints need to be placed on the CBoxes, negative capabilities are created to negate the usage of certain positive capabilities. This CBox is then attached to a process and the process is said to be confined within an environment created by the CBox.

There are several sources of capabilities. They are

- Regular Unix processes implicitly has all the capabilities associated with that process in the normal Unix sense.
- A regular Unix process which is root has more capabilities and could choose to delegate them.
- The capabilities could be packed in CBoxes as explained above and then be passed around.
- Executables can have capabilities associated with it which could be added dynamically to CBoxes.

Whenever the process tries to access any object, the applied CBox needs to contain the capability for that object. If the capability is unavailable with the CBox, access is denied. Negative capabilities cancels the positive one so the access is denied. When capabilities are available, its rights is checked and if it passes, the access is allowed. These

CBoxes are created in the kernel space and not in the user space. Hence, capabilities cannot be changed or forged or created illegally.

4.5.2 Locking CBoxes

A CBox which is in an unlocked state can be locked. If a CBox is locked the process is confined only to what the CBox has and no new capabilities could be added to it. In figure 4.1, the object and resource capabilities section is not locked and hence new capabilities can be added or dropped. If the section is locked then capabilities cannot be added but can be dropped for further confinement.

An important property of negative capabilities is that they can never be dropped irrespective of the CBox being locked or unlocked. This is an important property since the negative capabilities are used to apply important constraints and should not be dropped even if the negative sub section is in unlocked state. If the negative capabilities section is locked then negative capabilities cannot be added.

4.5.3 Modifying CBoxes

Once CBoxes are created, they could be modified dynamically. Modifications means that capabilities for objects can be added or dropped from the CBox. When a tighter environment is required, certain capabilities can be dropped. For example, capabilities for a particular user file can be dropped from the CBox so that the process can no longer access the file.

Capabilities can be added to CBoxes when required if they are not locked. This is done when access to certain objects is allowed dynamically. For example, a process attached to a CBox might require a capability to access files created by the user. Capabilities for this new file could be derived from the master and attached to the CBox so that the process could access the file. This feature also helps in realizing dynamic

changes in policies.

4.6 A Detailed Example

Let us explain using an example how capabilities can be used to create CBoxes and how programs can be confined within CBoxes to prevent any malicious activities. Let us consider a shell running in the Unix system. We want to successfully confine a mail program, *mail*, for that particular user. The mail executable has capabilities associated with it. These capabilities are derived from the master capabilities of mail program files and then attached to the CBox. For example, the capabilities for the file */etc/mail.rc* is added to the CBox with the pid of the process and userID of the user stamped on it. This capability has the execute permission for the file. Next, capabilities for the spool directory for the user are derived and added to the CBox. One of these files is */var/spool/mail/username*. Hence a read and a write capability is created for this file since read and write operations are required for the file while execution of the program. These capabilities are also stamped with the pid and the userID. Apart from these two files, one needs access to the */tmp* directory to create temporary files and use them later on. One might also require a resource capability to print the mail that the user has received. For example it could be a capability to print 50 pages and can be used like money.

Hence the CBox created for *mail* program is defined as

$$\begin{aligned} CBox(mail) = \quad & \{\langle deriv(mail.files)^{Stamp}\rangle, \langle deriv(resource)^{Stamp}\rangle, \\ & \langle deriv(spool.files)^{Stamp}\rangle, \langle (dummy.files)^{Stamp}\rangle\} \end{aligned}$$

Here $deriv(mail.files)$ refers to the collection of derived capabilities for the *mail* program. The set contains the capability for the file */etc/mail.rc* and a capability to create

files in the $/tmp$ directory. Next, $deriv(resource)$ refers to the derived capabilities of resources. In this case it contains a capability for printing a limited set of pages. It could be written as $(PrinterID, limit, Stamp, Lease, Properties)$. The $deriv(spool.files)$ refers to the derived capabilities for the spool files of the user. Two separate capabilities are required for read and write. They are $(FileID, read, Stamp, Lease, Properties)$ and $(FileID, write, Stamp, Lease, Properties)$. These capabilities are used separately for read and write operations respectively. The $(dummy.files)$ refers to a set of dummy capabilities attached to the CBox for creating capabilities for files created in the $/tmp$ directory.

After creating the CBox, the process is attached to it. When a process needs to access the mail program files and the spool files, it succeeds since the capabilities are available with the CBox. When the process tries to create a temporary files in the $/tmp$ directory, it is allowed since it has the capability to do so. Also, access to only those files created by the process is available and no other files present in the $/tmp$ directory since it does not have the capability for those files. Capability for creating files in $/tmp$ directory does not mean that it can access all files in the directory. This achieves confinement as well as conforms with principle of least privileges.

When the process needs to print the mail, it can use the resource capability to do so. Once it uses the capability, the limit for the number of pages it can print is reduced by the number of pages it has printed.

Now, when the mail program tries to access any other files or objects, it wont be able to do so since it does not have the capability for them.

The mail program might need to execute a *jpeg viewer* in order to display the image attached to the mail that it receives. Hence, when the mail process forks, the jpeg viewer is executed and the capabilities associated with it are added to the CBox. Figure 4.2 illustrates the state of the CBox before and after the process forks. As shown in the

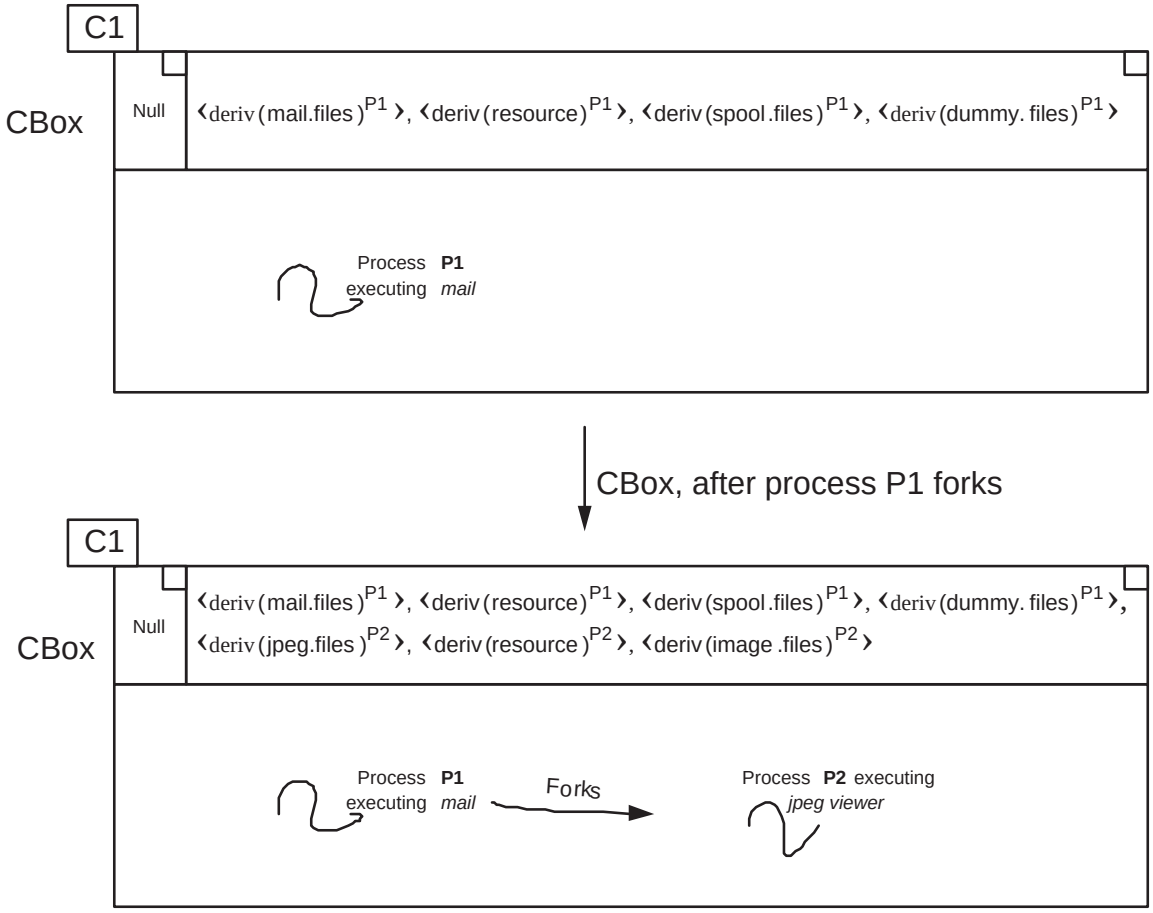


Figure 4.2: Example describing CBoxes

diagram when the newly added derived capabilities are stamped with the pid of the new process and hence can be accessed only by it and not P1. Also, P2 cannot access the mail files since the derived capabilities for mails files are stamped with pid P1. P2 can use the capabilities for *jpegviewer* files to execute the program. It can access the image files using $deriv(image.files)$. If the jpeg image needs to be printed it can use the resource capability stamped with P2 to do so.

This way, CBox is changed dynamically and various programs that are executed can be confined and provide security. The next chapter illustrates the implementation details of the mechanism on a linux kernel.

Chapter 5

Implementation on a Linux Kernel

The Linux Operating System is used as the platform to test the effectiveness of the capability based mechanism. We use the Linux Security Module(LSM) [Wri02] as the access control framework which enables many different access control models to be implemented as loadable kernel modules. SELinux [Los01] and Domain and Type Enforcement [Lee96] have already been adapted to use the LSM framework. We choose LSM because of the following reasons:

- The generality of LSM permits enhanced access controls to be effectively implemented without requiring kernel patches.
- Truly generic, where using a different security model is merely a matter of loading a different kernel module.
- Conceptually simple, minimally invasive, and efficient.
- No kernel modifications are required to overwrite entries in the system call lookup table.

- LSM allows modules to mediate access to kernel objects by placing hooks in the kernel code just ahead of the access.

In the next section we describe LSM in brief to provide a context for describing the capability based system.

5.1 LSM - Linux Security Modules

LSM takes the approach of mediating access to the kernel's internal objects: tasks, inodes, open files, etc., as shown in Figure 5.1. User processes execute system calls, which first traverse the Linux kernel's existing logic for finding and allocating resources, performing error checking, and passing the classical UNIX discretionary access controls. Just before the kernel attempts to access the internal object, an LSM hook makes an out-call to the module posing the question, "Is this access ok with you?" The module processes this policy question and returns either "yes" or "no."

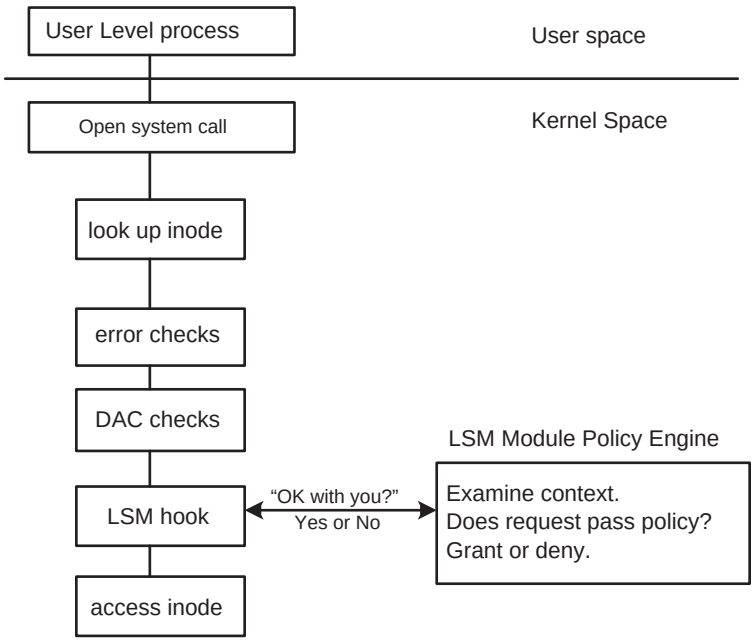


Figure 5.1: LSM Hook Architecture

The basic abstraction of the LSM interface is to mediate access to internal kernel

objects. LSM seeks to allow modules to answer the question "May a subject S perform a kernel operation OP on an internal kernel object OBJ?"

The LSM kernel patch modifies the kernel in five primary ways. First, it adds opaque security fields to certain kernel data structures. Second, the patch inserts calls to security hook functions at various points within the kernel code. Third, the patch adds a generic security system call. Fourth, the patch provides functions to allow kernel modules to register and unregister themselves as security modules. Finally, the patch moves most of the capabilities logic into an optional security module.

There are various hooks available in LSM to mediate various structures which forms an integral part of the kernel and security. For example, program loading hooks include the *binprm_*

security_ops, to manage the process of loading new programs. LSM adds a security field to the *linux_binprm* structure. File system hooks include hooks for mediating objects like files and directories. LSM adds a security field to each of the associated kernel data structures: super block, inode, and file. Several other hooks are also available for network, IPC and module hooks.

5.2 Design of Capability Based Mechanism using LSM

As mentioned earlier, we make use of the LSM hooks to mediate the access to an object. For achieving a fine grained access control, we have defined several access rights for each type of object. As shown in table 5.1 the object 'file' has not only the conventional 'read', 'write', 'execute' permissions but also other permissions like 'create', 'open', etc. Similarly, other objects like 'dir', 'socket' and 'ipc' also have fine grained permissions.

Table 5.1: Objects and Fine Grained Permissions in CBox

OBJECT	PERMISSIONS
file	execute, unlink, setattr, quotaon, relabelfrom, link, write, ioctl, relabelto, read, rename, append, lock, swapon, getattr, removeattr, mounton, create, execute_no_trans, entrypoint
dir	create, read, write, execute, link, unlink, rename, append, setattr, getattr, removeattr, ioctl, search, rmdir, mount, lock, swapon, quotaon, relabelfrom, relabelto, add_name, remove_name, reparent
socket	relabelto, recvmsg, relabelfrom, setopt, append, setattr, sendto, getopt, read, shutdown, listen, bind, write, accept, connect, lock, ioctl, create, namebind, sendmsg, recvfrom, getattr, connectfrom, connectto, sendfrom, node_bind
file system	mount, remount, unmount, getattr, relabelfrom, relabelto, transition, associate, quotamod, quotaget
ipc	sem_setattr, sem_read, sem_associate, sem_destroy, sem_unix_write, sem_create, sem_unix_read, sem_getattr, sem_write, msgq_setattr, msgq_read, msgq_associate, msgq_destroy, msgq_unix_write, msgq_create, msgq_unix_read, msgq_getattr, msgq_write,msgq_enqueue, msg_send, msg_receive, ipc_create, ipc_destroy, ipc_getattr,ipc_setattr,ipc_read, ipc_write, ipc_associate, ipc_unix_read, ipc_unix_write
process	fork, transition, sigchld, sigkill,sigstop, signull, signal, ptrace,getsched,setsched, getsession, getpgid, setpgid, getcap, setcap, share, getattr, setexec, noatsecure, siginh, setrlimit, rlimitinh, file_receive

5.2.1 Opaque Security Fields - Master Capabilities

The opaque security fields are void* pointers, which enable security modules to associate security information with kernel objects. Table 5.2 shows the kernel data structures that are modified by the LSM kernel patch and the corresponding abstract object. These opaque security fields are used to store the master capability for the objects. Separate structures are defined for master capabilities of different objects. The master capability structure of inode is shown in figure 5.3. The master capability contains an integer field for 'ObjID' and the 'type' field to identify the

Table 5.2: Kernel data structures modified by the LSM kernel patch and the corresponding abstract objects.

STRUCTURE	OBJECT
task_struct	Task(Process)
linux_binprm	Program
superblock	Filesystem
inode	Pipe, File or Socket
sk_buff	Network Buffer (Packet)
net_device	Network Device
kern_ipc_perm	Semaphore, Shared Memory Segment, or Message Queue
msg_msg	Individual Message

```
int vfs_create(struct inode *dir, struct dentry *dentry, int mode, struct nameidata *nd)
{
        int error = may_create(dir, dentry, nd);

        if (error)
                return error;

       if (!dir->i_op || !dir->i_op->create)
                return -EACCES;
        mode &= S_IALLUGO;
        mode |= S_IFREG;
<->     error = security_inode_create(dir, dentry, mode);
        if (error)
               return error;
       DQUOT_INIT(dir);
       error = dir->i_op->create(dir, dentry, mode, nd);
        if (!error) {
                inode_dir_notify(dir, DN_CREATE);
<->             security_inode_post_create(dir, dentry, mode);
        }
        return error;
}
```

Figure 5.2: The `vfs_create` kernel function with one security hook call to mediate access and one security hook call to manage the security field. The security hooks are marked by <->

type of object. For most kinds of objects, an `alloc_security_hook` and a `free_security_hook` are defined that permit the security module to allocate and free security data (master capability) when the corresponding kernel data structure is allocated and freed. Other hooks are provided to permit the security module to update the security data as necessary, e.g. a `post_lookup_hook` that can be used to set security data for an inode after a successful lookup operation.

5.2.2 Calls to Security Hook Functions

LSM provides a set of calls to security hooks to manage the security fields of kernel objects. It also provides a set of calls to security hooks to mediate access to these objects. Both sets of hook functions are called via function pointers in a global security ops table. This structure consists of a collection of substructures that group related hooks based on kernel object or subsystem, as well as some top-level hooks for system operations.

Figure 5.2 shows the `vfs_create` kernel function after the LSM kernel patch has been applied. This kernel function is used to create new inodes. Two calls to security hook functions have

been inserted into this function. The first hook call, `security_inode_create(dir, dentry, mode)`, is used to control the ability to create new inodes. If the hook returns an error status, then the new inode will not be created and the error status will be propagated to the caller. This hook is used to check whether the current process has the capability to access the object and whether the capability is valid or not. The implemented code for this hook is as shown in Figure A.1 of Appendix A. The general algorithm which is used to implement is illustrated in Table 5.3 and needs to be tailored for specific type of capabilities. The second hook call, `security_inode_post_create(dir, dentry, mode)`, can be used to set the security field for the new inode structure. This hook is used to update the master capability for the inode created as explained earlier. The implementation details are shown in Figure A.1 of Appendix A.

Table 5.3: Algorithm for Access Hook

1. Check whether the current process requesting the access is attached to a CBox.

2. If it is not, then access is allowed since the process is not confined to any CBox and is thus controlled by the Unix security mechanism.

3. If the process is attached to a CBox do the following -

3.1 Check whether CBox contains the capability for the object with the permission. If its not, access is denied, else goto next step

3.2 Check whether the PID of the capability matches with the PID of process. If it does not, return error "Capability Not Valid", else goto next step.

3.3 Check whether there is any negative capability for the object. If there is, return error "Capability Invalid", else goto next step.

3.4 Access Allowed. Return 0.

4. End hook.

Similar hooks were written for other types of objects like IPC, Filesystem, Network etc. Now we discuss how other types of capabilities are created and the implementation details of CBox.

5.2.3 Creation of Other Capabilities and CBoxes

Separate structure are defined for master, derived, resource and encrypted capabilities. The derived capability structure has five fields and each field has appropriate types set. Different

structures for capabilities are defined for different types of objects. The structure of file capabilities is depicted in Figure 5.3.

Master Capability attached to Inode

```
struct cap_struct_master_inode {

   unsigned long i_no;        /* Inode Number*/
   unsigned int type;        /* type of object */
   enum cbox_file_permissions perm;
   struct inode *inode;      /* back pointer  */
};
```

Encrypted Capability Structure

```
struct cap_struct_encrypted_file {

unsigned long i_no;
unsigned int type;
enum cbox_file_permissions perm;
char code[ ];                    /* Encrypted Code */

};
```

Derived Capability attached to CBox

```
struct cap_struct_derived_file {

 unsigned long i_no;
 enum cbox_file_permissions perm;
 unsigned int type;
 unsigned int pid;
 unsigned int uid;
 unsigned int derivable;       /* Properties */
 unsigned long lease;          /* Expiry Time */
 };
```

Resource Capability attached to CBox

```
struct cap_struct_resource {

unsigned int ResourceID;
unsigned int type;
unsigned int limit

};
```

Figure 5.3: Capability Structures

For CBoxes, we define a structure which has a list of derived, negative, resource and dummy capabilities. The CBox structure has an integer field for CID. The structure of CBox is as shown in Figure 5.4

```
struct cbox_struct {

unsigned int cid;                                   /* CBox ID. */
unsigned int size;                                        /*Max No. of Capabilities */
struct cbox_capability_derived *deriv_list;          /*List of Capabilities in CBox.*/
struct cbox_capability_negative *negative_list;    /* Negative Capabilities */
struct cbox_capability_resource *resource_list;    /* Resource Capabilities */
struct cbox_capability_dummy *dummy_list;          /* Dummy Capabilities */
unsigned int lock;
struct cbox_struct_list *cbox_list;
}
```

Figure 5.4: CBox Structures

As defined earlier, processes need to use capabilities if they are in a CBox. When a process does an exec, a new CBox structure is allocated and the capabilities associated with the exec

are attached to the CBox. The process is then attached to the CBox by adding the CID to the security structure of the process created by LSM. To access an object the algorithm shown in table 5.3 is used. This algorithm describes how we implement all the features of the capabilities including positive and negative capabilities. This basic algorithm applies for all the accesses performed through the CBox.

5.3 Performance Evaluation

We have implemented the capability based mechanism in a Red Hat Professional Workstation with the linux kernel version 2.6.9. The Linux kernel runs on a Pentium-4 3.2GHz system with 512MB RAM.

5.3.1 Microbenchmark - Test programs and Results

To evaluate the performance overhead introduced by the capability checking, microbenchmarks were performed, which uses the open-file program and the fork program found in the Appendix B. The open-file program is used to test the overhead introduced by the file capability attached to the process used for access control. Here, the process accessing the file is attached to a CBox and the capability for that file is contained in the CBox. This program tries to open a file 1000000 times and the total time taken is as shown. Table 5.4 illustrates the overhead incurred while using capabilities in CBox to access the file and without using the capability.

The fork program is used to test the overhead introduced when a new CBox is created for the forked process. This program tries to fork 10000 times. Every time it forks, a new CBox structure is allocated and capabilities attached to it. The child process exits immediately and the CBox structure is freed. Hence the time taken to create a CBox leads to the extra overhead shown in table 5.4.

5.3.2 Using LMBench for microbenchmarking

We also use LMBench [McV96] as our microbenchmark. LMBench was developed specifically to measure the performance of core kernel system calls and facilities, such as file access, context switching, and memory access. LMBench has been particularly effective at establishing and

Table 5.4: Microbenchmark for open() and fork() in seconds

	Open File			fork		
	User	**System**	**Real**	**User**	**System**	**Real**
2.6.9	0.13	0.67	0.81	0.15	0.52	0.68
2.6.9-lsm	0.13	0.67	0.82	0.15	0.52	0.69
% Overhead	0%	0%	1.2%	0%	0%	1.4%
2.6.9-CBox	0.135	0.69	0.85	0.16	0.55	0.70
%Overhead	3.8%	2.9%	3.65%	6.67%	5.7%	2.9%

maintaining excellent performance in these core facilities in the Linux kernel.

We use LMBench to measure the time taken for creating and removing files. The files are created in a particular directory. Initially LMBench is run without CBoxes and the values are recorded. Later, the same process is attached to a CBox and capabilities are attached to the CBox for creating files in the directory. On successful creation of the file, a master capability is created and attached to the file. When the file is deleted the master capability structure is freed. The LSM framework imposes minimal overhead when compared with a standard Linux kernel. We compared a standard 2.6.9 kernel with and without LSM. Then, we determined the time taken with CBox attached to the process. The overhead as shown in table 5.5 is around 2-3% which is negligible and can be improved further by efficient implementation.

Table 5.5: File and VM system latencies in microseconds - smaller is better

Test Type	**2.6.9**	**2.6.9-lsm**	**% Overhead**	**2.6.9-CBox**	**% Overhead**
0K file create	24.1	24.2	0.4%	24.8	2.9%
0K file delete	3.70	3.70	0%	3.70	0%
10K file create	56.2	56.8	1%	58.0	3.2%
10K file delete	10.8	10.9	0.9%	11.0	0.9%

5.3.3 Macrobenchmark

For macrobenchmark, we ran an "untar linux kernel source" script. The "untar linux kernel source" script is used to test the overhead on a general application. The script will untar a compressed linux source tarball (linux-2.6.9.tar), tar the source tree again, finally remove the tarball and the source tree. The shell script written is shown in Appendix B. Initially the script was executed with the tar process not attached to any CBox. The CBox was then attached to the process and appropriate capabilities added. The results of the overhead are shown in table

5.6

Table 5.6: Macrobenchmark using *tar*

Kernel	**Untar Kernel Source**		
	User	**System**	**Real**
linux-2.6.9	0.28	1.88	27.57
linux-2.6.9-lsm	0.28	1.88	27.57
% Overhead	0%	0%	0%
linux-2.6.9-cbox	0.28	1.94	28.61
% Overhead	0%	3.2%	3.77%

The overhead as shown in the table is around 3% which is not too high. To summarize, preliminary prototype implementation shows that overheads leavel are small given the additional security. A more sophisticated and optimized implementation we believe will reduce it to practically negligible as the current prototype is unoptimized.

Chapter 6

Using Capabilities to Implement RBAC

This chapter shows the flexibility of the capability mechanism . RBAC is a general scheme for providing least privilege. We show how our capability system can be used as a mechanism for implementing RBAC. Each section discusses how various parts of the RBAC model can be modeled using capabilities discussed in the previous chapters. The following work has been submitted as a conference paper [Yap05] and awaiting the result.

6.1 A Capability based RBAC

The capability mechanism can be used to implement the core components of RBAC. Capabilities are used directly to provide the permissions associated with roles and also enable the inheritance of permissions in roles. Switching of roles in sessions and session locking also use features of the capability system. Capabilities are also used as keys (tokens) to enable authentication for operations on roles.

6.1.1 Implementing the Core RBAC

In this section, for simplicity, we will consider the administration of RBAC and operations on RBAC to be managed by a single *role manager*. The simplest role manager would be a distin-

guished process which already has all the capabilities it needs for all role permissions. Rather than a superuser style role manager, it is also possible for the role manager to be unprivileged and acquire the appropriate capabilities. For example, the human administrator could pass to the role manager the capabilities needed in a demand-driven fashion. For now, we will assume that the role manager already has capabilities for all the basic permissions needed in all roles.

Roles are viewed as objects which are managed by the role manager. A role is an object with a master capability and the following fields:

- keys of children roles (*child* field): these are derived capabilities which serve as authorization keys for a child role.
- permissions of this role (*perm* field): this is a container capability which consists of two sub-containers, $perm^+$ for those capabilities which give the access permissions and $perm^-$ which are used to impose constraints. The containers themselves may either be locked or not depending on how much power is given to the role manager to manipulate permissions.
- other fields such as users, pointers, etc. which are required to manage the role hierarchy and RBAC. We remark that the user role assignments (URA) in RBAC is simply the correct use and maintenance of the user fields in the role objects by the role manager. We focus in this thesis on how capabilities can be used to operate on roles and maintain the permissions.

Permissions are granted by deriving capabilities from master capability of the original object. Derived capabilities are used in sessions so that permissions can only be further restricted rather than being arbitrarily changed.

Consider the role R1 in the role hierarchy in Figure 6.2. The RBAC permissions for a user U1 in session S1 is modeled as follows as depicted in Figure 6.1. The R3 role object has: (i) child capabilities for R2, $deriv(R2)$, and R3, $deriv(R3)$; and permissions capability container $R1.perm$. For user U1, a capability $deriv(R1.perm)^{U1}$ which is stamped with U1 serves as U1's role permissions in all sessions with role R1. A session S1 depicted in Figure 6.1 is created as follows:

- The R1 capability for U1 is added to the environment, $Env(S1) = \langle deriv(R1.perm)^{U1,S1} \rangle$

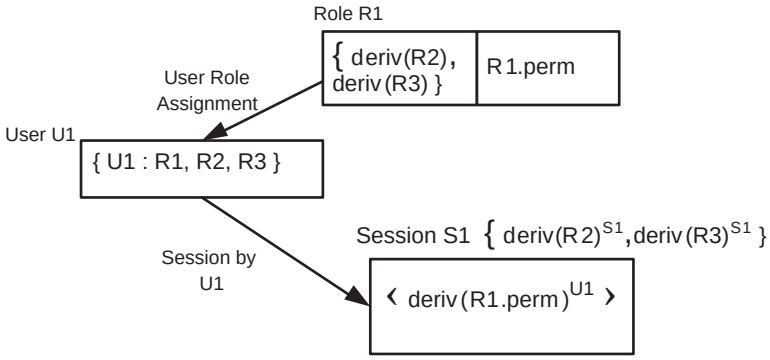


Figure 6.1: Example of a role and session in core RBAC

and is additionally stamped with S1.

- The derived child role capabilities are stamped with S1, $deriv(R2)^{S1}$ and $deriv(R3)^{S1}$, are given to S1. Note that these are not added to the environment and are meant to be used as authorization keys.
- The current role(s) for the session is set to R1.

Since the permission capability in S1 is stamped with U1, no other user can make use of it. This however is not strictly needed because capability $deriv(R1.perm)^{U1}$ is already in S1's environment and hence cannot be extracted. The stamping here is useful for revocation. The child capabilities are stamped with S1, these capabilities are meant as keys and not as permissions to access the role object and the stamping serves to uniquely identify the session and roles.

The permissions of R1 are contained in its capability container which is already in the environment so S1 has exactly the permissions it needs. If S1 can obtain more capabilities, this can only occur by the use of its existing capabilities. It is also possible to ensure that S1 cannot acquire any more capabilities beyond its role R1 by locking the environment. We however do allow S1 to further restrict permissions by deleting positive capabilities or by adding negative capabilities to its environment.

6.1.2 Hierarchical RBAC using Capabilities

The RBAC model organizes roles into hierarchies using the $\succeq$ relation for role inheritance. A role $r_1 \succeq r_2$ if the permissions of r_2 are a subset of r_1, $r1.perm \succeq_c r2.perm$, and the users of r_1

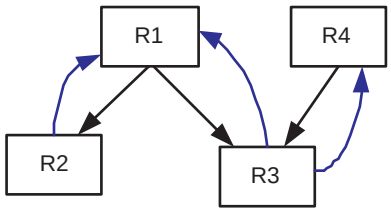


Figure 6.2: Example Role Hierarchy

are a subset of the users of r_2.

The role hierarchy can be represented by a directed acyclic graph induced by role inheritance. The role objects mimic the DAG structure because the derived child record the immediate sub-roles of any role. In Figure 6.2, the downward arrows show the role inheritance while the upward arrows show the derived child capabilities.

There are a number of ways to model the effect of the role inheritance on the permissions. Assuming that the role manager has access to capabilities representing individual permissions, the permissions of each role can be modeled using a master container capability with the appropriate capabilities inserted to represent the permission role assignment (PA) relation.

Suppose the role hierarchy is a collection of trees. One could view the permissions of an immediate sub-role to be a restriction of the permissions of its parent role. Suppose role $R1$ is a parent of role $R2$. The permission of $R2$ can simply be a container capability with $deriv(R1.perm)$ inserted where the role manager has removed the capabilities in $R2$ which are not in $R1$. If the role manager changes the permissions of $R1$, e.g. a capability is removed from $R1.perm$ (note that $R1.perm$ could itself be a derived capability), the effect is propagated to $R2.perm$ by virtue of it containing a derived capability from $R1$. We will call this use of the capability system, *derived role permissions*. It is still possible to change $R2.perm$ since it is a container, so the current $deriv(R1.perm)$ could be deleted and a new one with different restrictions inserted as $deriv(R1.perm)'$.

In RBAC, the parent role is considered to inherit the permissions of the child roles. However, with multiple inheritance, this would lead to roles which are incomparable in the hierarchy inheriting permissions from each other. Using Figure 6.2 as our reference example, role R3 is contained in both roles R1 and R4. This means that because of upward inheritance of per-

missions R1 can gain permissions of R4 which are in R3 (and vice versa). Capabilities allow a stricter interpretation that a parent role R1 can assume the permissions of a child role like R3 without inheriting the permissions of R3. By using derived role permissions in the presence of multiple inheritance, we can make $R1.perm$ incomparable with $R3.perm$ rather than $R1.perm \succeq_c R3.perm$. The $R3.perm$ container consists of $\langle deriv(R1.perm), deriv(R4.perm)\rangle$. A session which assumes roles R1 and R3 will have the original RBAC definition of upward permission inheritance since its environment will contain

$$\langle R1.perm, \langle deriv(R1.perm), deriv(R4.perm)\rangle\rangle.$$

6.1.3 User Sessions

User sessions can be created to assume one or more active roles. This gives the user session the permissions of its active roles. A new user session is created with the help of the role manager to have an environment containing the capabilities of all its active roles. In the multiple inheritance example earlier, the session having activated roles R1 and R3 has both capabilities in its environment.

An existing session can choose to further restrict its active roles to sub-roles of those active roles. Changing roles in this fashion is a voluntary operation on the part of the session which serves to reduce its existing privileges. Thus, the only issue is to ensure the environment is correctly set up. As the session already has capability keys for its immediate child roles, it just has to present the appropriate ancestor key(s) of the role(s) to switch to. Because the keys are stamped to the current session, only this session can have access to the keys. Even if the session attempts to leak the capabilities obtained by the role manager to another session, it would be useless since the session stamp on the capabilities would make them invalid.

The new capabilities for the new roles can be added to the environment and the old ones deleted. Locking the environment of the session serves to stop any changes to the roles of a session.

6.1.4 Constraints using Capabilities

There are two kinds of constraints in RBAC: static (SSD) and dynamic (DSD) separation of duties. SSD prevents a user from being a member of one or more roles. Another way of looking at this for a binary static separation constraint is that two roles are mutually exclusive.

While any constraint can simply be enforced by the correct behavior from the role manager, it is desirable to be able to ensure that the underlying security mechanism be able to also enforce this. Consider two roles R1 and R2 which are in a SSD constraint. We can attach to R1 (respectively R2) a negative capability to its permissions. So $R1.perm$ (respectively $R2.perm$) has the capability $deriv-(R2.perm)$ added (respectively $deriv-(R1.perm)$). Now even if a user was incorrectly added to both roles R1 and R2, the environment with the two capabilities would cancel out leading to no permissions at all. One subtlety is that when derived role permissions are used, the negative capabilities cannot be attached directly to the permissions but rather to the derived capabilities which are to be used in the environment. We remark that because SSD is inherited upwards, the same process is used for ancestor roles as well.

DSD cannot be directly enforced by capabilities as it requires taking into account the history of a session. However the role manager can assist in the enforcement. Essentially DSD is a mutual exclusion requirement placed on role activation within a session. Consider role R1 which cannot be activated together with role R2. When role R1 is activated by user U1, the role manager adds $deriv^-(R2)$ to its derived capability for this role which it will use for sessions from U1 and also $deriv^-(R2)$ to the corresponding capability for R2 for U1.[1] This situation is one where the DSD has scope across all sessions of U1. After role R1 is invoked, the capabilities from role R2 can never be used in any further sessions of U1. The DSD scope can be restricted to just the session itself by the role manager maintaining the derived capability on both a user and session basis.

[1]The use of a $deriv^-(R2)$ is similar to revoking the capability but saves having to remember that R2 cannot be used further.

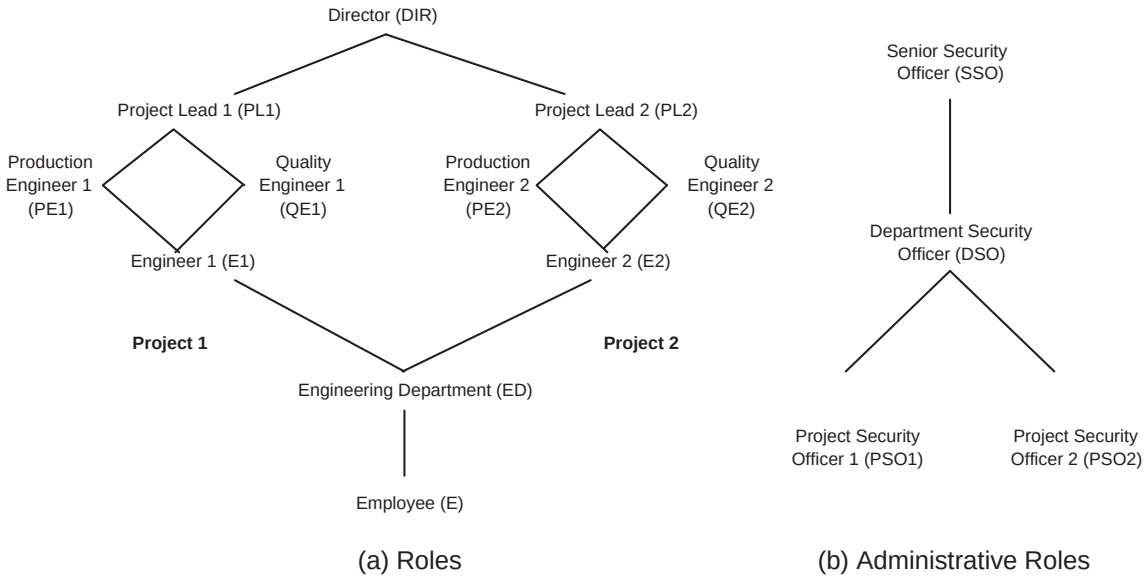


Figure 6.3: Distributed Administration Example

6.1.5 Dynamic RBAC using Capabilities

Implementing RBAC with capabilities allows more fine grained control than vanilla RBAC. In RBAC, a session can reduce its privileges by switching to a sub-role. While a role does achieve a form of least privilege, the privileges to which a role has may still be more than needed simply because a role is pre-defined statically by the role administrator and has to cater for a general instance of the role.

The capability system here allows the session to determine dynamically how to reduce its own privileges by either deleting capabilities from its own environment, adding negative capabilities or by replacing its capabilities with further derived ones. Even if the environment is locked, the session is still allowed to further restrict its existing capabilities.

6.2 Administrating Capability-based RBAC

The capability-based RBAC gives some advantages in administration in the handling of permissions and in the deletion of roles. Permissions in the form of capability containers allows the packaging of permissions with arbitrary granularity inside a container. Revoking a container capability allows in a single operation to remove all its permissions. Conversely, adding a container

to a role adds its set of positive and negative capabilities. This can simplify administration by organizing sets of permissions together. The delegation example below illustrates this use of containers as atomic building blocks.

The use of derived role permissions can simplify the management of permissions since permission changes on a parent role propagate to its children. While this means that the role manager is restricted to making only certain changes, this restriction could be a good thing (note that it is not necessary to use derived role permissions). For example, consider the role hierarchy in Figure 6.3. Suppose the goal is have any changes to permissions to an employee E be automatically reflected in all higher roles. Furthermore, changes in the project file access for PL1 are to be reflected to PE1, QE1 and E1 modulo the restrictions on permissions in those roles. The basic permissions of E can be encapsulated in a container capability which is derived to all the other roles The project file access permission for PL1 are encapsulated in a sub-container which is derived and added to the permission containers for roles PE1, QE1 and E1. In this example, the derived role permission flow is in both upward and downward directions in the hierarchy with respect to permission changes to E and PL1.

Deleting a user from a role is just a matter of revoking the derived capability which is in the active user sessions. Since the capability in the environment of active sessions for that role is now unusable, the sessions no longer have the permissions of that role. A similar approach to DSD constraints can be used to delete the user from sub-roles if desired.

Deleting a role similarly also revokes the capabilities from that role object. When derived role permission is used in the sub-roles, then the permissions from sub-roles which derived from the deleted role are also revoked. This saves some management on the part of the role manager and also ensures that the effect on sessions from the deletion is effective upon deletion.

6.2.1 Delegation of Administration in RBAC

Capabilities can be used for decentralized administration by delegating some of the permissions for maintaining RBAC to several role managers. Consider several role managers organized in an administrative hierarchy. For example in Figure 6.3 the delegation is as follows: the administrative scope of PS01 is PL1, PE1, QE1 and E1; DSO administers PL1 and PL2; and

SSO administers DIR, ED and E. DSO can give PS01 the derived permission capabilities for PL1. Thus PS01 can only further restrict those permissions. Furthermore, the role objects for PL1, PE1, QE1 and E1 could already be created with the derived capabilities also given to PS01. PS01 may be operating in a restricted and locked environment so that pretty much all it can do is to refine the permission assignment to the roles that it is administrating or to change user role assignments. Further restrictions on user role assignments by only giving say the engineering group a stamped capability which allows these users to communicate with PS01. Thus administration can be distributed but the power of the delegated administration can still be restricted.

DSO is allowed to change the permissions of PS01 within his own restrictions imposed by SSO. Only SSO the capabilities for the role objects of DIR, ED and E, so only SSO can change them.

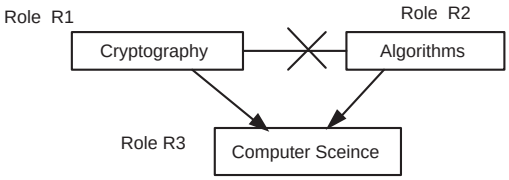
6.3 Some Examples

In this section, we will illustrate a simple complete example of RBAC and also an extension of the example to delegation of administration. The example in Figure 6.4 considers a Computer Science department with two courses Cryptography and Algorithms with the hierarchy given in 6.4(a). The roles here concern the six objects O1 to O6 each with its own master capability, e.g. (O1, ALL) is the master capability with all rights to O1. An object could be a file or another resource such as a printer, scanner, etc.

The permission role assignment gives the roles access to the objects using derived capabilities. For example, role R1 has read access to the file `crypto_assignment.pdf` A special limit capability is illustrated in O6 for role R3, this could for example restricted how many pages can be printed on a printer.

The child hierarchy is illustrated with the $deriv(R3)$ capability in R1 and R2. There is a SSD which arises because students taking Cryptography are not allowed to take Algorithms and vice versa. This may be due to a change in the syllabus requirements. This gives rise to a negative capability, $(R2, Negative)$ in the permissions container of R1.

Students Bob and Alice are assigned to roles R1 and R2 respectively which are their courses



(a) Role Hierarchy

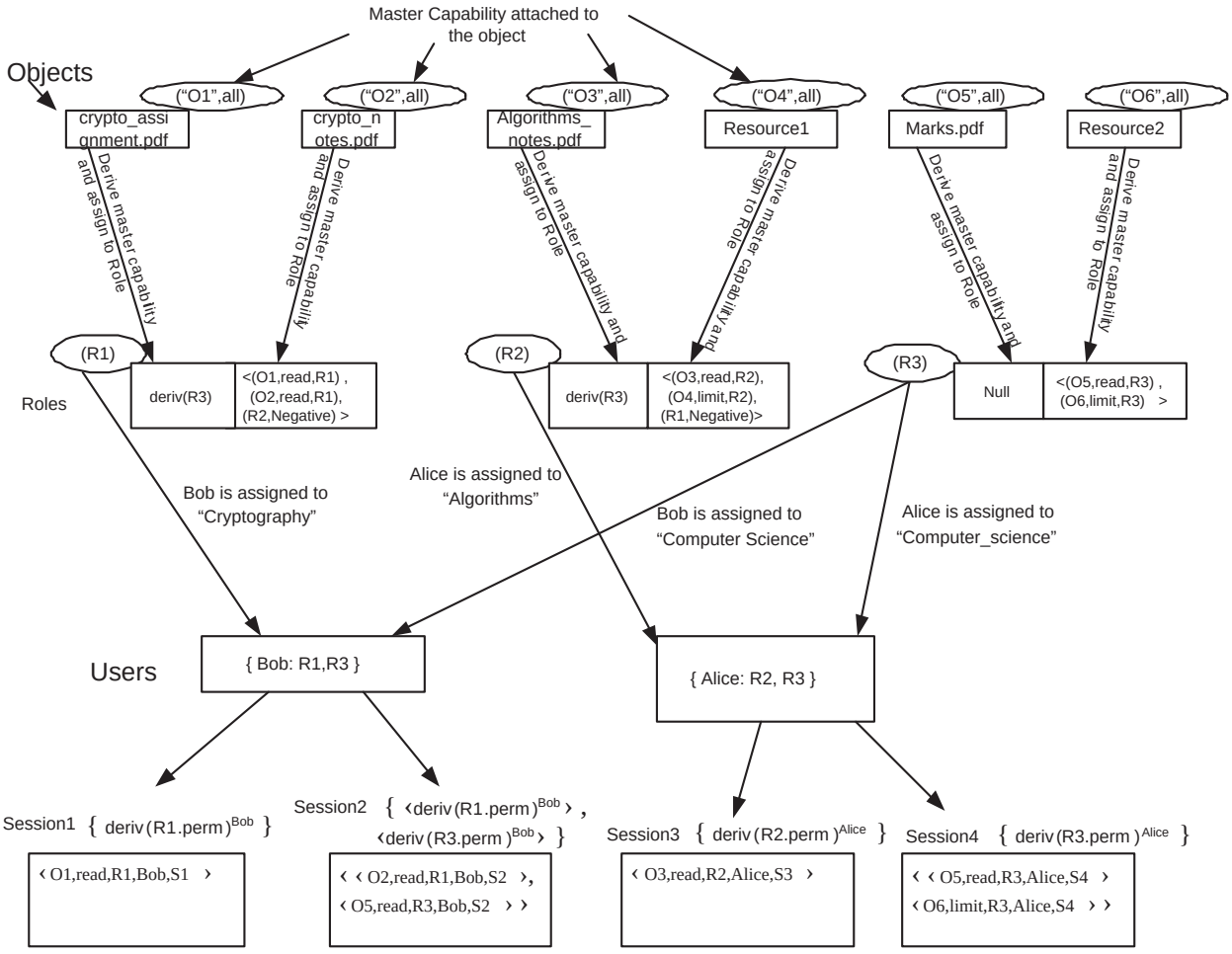
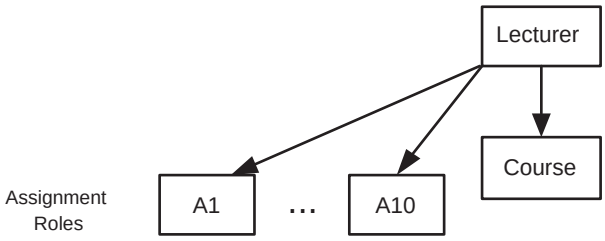
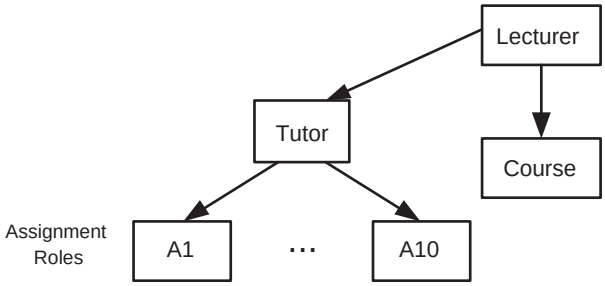


Figure 6.4: Example describing usage of capabilities for RBAC



(a) Role Hierarchy before Delegation



(b) Role Hierarchy after Delegation

Figure 6.5: Delegation Example

for the current semester. They are also implicitly a user to role R3.

In this example, the role permissions do not follow strictly the role inheritance of normal RBAC to illustrate that rbac-like models can be created. Here, we want a CS student has some basic permissions while a course student can access both the CS role and the course role but the course permissions do not necessarily contain the permissions in the CS role.

Bob creates session 1 activating only R1. Session 2 activates R1 and R3 but has further restriction beyond the role permissions. Here its environment only contains permissions for O2 and O6. This could be achieved either by the role manager or Bob himself removing capabilities for O1 and O5. The capability O6 in session 2 is a resource limited one so one could view session 2 as one specially created just for printing the `crypto_notes.pdf` and the limit resource capability could restrict the maximum number of pages printed per session.

Now consider the Cryptography course where students can be given access to particular assignments A1, A2, ... Different students may have access to different assignments. One way to do so is that a lecturer role assigns a user to the student role and the chosen set of assignment roles as shown in Figure 6.5. A lecturer would rather delegate this task to a tutor and gives

him the container capability with all the permissions for roles A1, A2, ... The tutor also gets a role manager for the assignments. So the tutor can now assign students to their respective assignment roles independently of the lecturer.

Chapter 7

Conclusion and Future Work

This capability based mechanism has several advantages. Its a dynamic scheme and provides flexibility in terms of managing the capabilities and using it for access control. For propagation, capabilities need to be derived and attached to CBoxes. To demonstrate the power and flexibility, the capability based mechanism can also be used to implement all the aspects of RBAC including hierarchy, constraints and administration. CBoxes can be easily created by users and the derived capability is the proof of possessing a right to access the object. This capability based mechanism helps create a powerful permission model. Use of special capabilities like one-time capabilities can provide more restriction. Dropping of capabilities can provide revocation dynamically. Leasing can also be introduced where capabilities are leased for a certain amount of time providing time dependent capabilities.

We obtain confinement using capabilities by creating closed domains for programs to run in. The principle of least privileges forms an important basis of using capabilities. The all powerful root is eliminated in our system. Hence, even if a malicious user gets access of the root account, he cannot do much since root can be confined. Users gets the option to create a dynamic CBox to restrict untrusted programs. This in a way is a combination of MAC and DAC. Fine grained capabilities are created for fine grained permissions.

Future work could include using this capability based mechanism in distributed operating systems and networks. This would require introduction of a few more types of capabilities. Distributed Capability based systems could be similar to the one proposed in Amoeba [Tan86]

with a more powerful permission model using the capabilities.

Bibliography

[Ach99] Anurag Acharya and Mandar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Application. *Dept. of Computer Science, University of California, Santa Barbara.* 1999.

[Ale99] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations. *In USENIX Annual Technical Conference* 1999.

[Bib96] M. Bishop and M. Dilger, Checking for Race Conditions in File Accesses, *Computing Systems 9 (2) pp. 131-152* Spring 1996.

[BsdJa] FreeBSD jail manual page. JAIL(2) *http://www.freebsd.org/cgi/man.cgi?query=jail&sektion=2*

[Coh75] Ellis Cohen and David Jefferson. Protection in the Hydra Operating System, *Proceedings of the fifth ACM symposium on Operating systems principles,pg 141–160* 1975.

[Dat89] Datapro and Bull HN. Information Systems Inc: Security Capabilities of Multics, *USA: Datapro Research; IS56-115-101. Datapro Reports on Information Security; Vol 3*, April 1989.

[Dav01] David Ferraiolo, Ravi Sandhu, Seban Gavrila, D. Richard Kuhn, Ramaswamy Chandramouli. Proposed NIST Standard for Role-Based Access Control. *ACM Transactions on Information and System Security, Vol. 4, No. 3,* August 2001, 224274.

[Den65] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations *MIT/LCS/TR-23* 1965.

[Gol96] Ian Goldberg, David Wagner, Randi Thomas, Eric A. Brewer. A Secure Environment

for Untrusted Helper Applications Confining the Wily Hacker. USENIX Security Symposium, 1996.

[Gon89] L. Gong. A Secure Identity-Based Capability System. *Proceedings of the 1989 IEEE Symposium on Security and Privacy, p. 5665.* 1989.

[Har85] N.Hardy. Keykos architecture. *SIGOPS Oper. Syst. Rev., 19(4):825,* 1985.

[Ian96] Ian Goldberg, David Wagner, Randi Thomas and Eric A. Brewer A Secure Environment for Untrusted Helper Applications, *Proceedings of the 6th Usenix Security Symposium, San Jose, CA, USA,* 1996.

[Kao96] L.Kao and R.Chow. An extended capability architecture to enforce dynamic access control policies. *In ACSAC, pages148157,* 1996.

[Kar03] A. H. Karp, R. Gupta, G. J. Rozas, A. Banerji. Using Split Capabilities for Access Control. *IEEE Software 20(1),* January 2003, p. 4249.

[Lam73] B.W. Lampson. A Note on the Confinement Problem *CACM on Operating Systems, Vol.16, No.10* October, 1973.

[Lee96] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghighat. A Domain and Type Enforcement UNIX Prototype. *In Proceedings of the 5th USENIX Security Symposium.* 1996.

[Lev84] H. M. Levy. Capability-Based Computer Systems. *Digital Press,* 1984.

[Los98] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrell. The Inevitability of Failure: The Flawed Assumption of Security in Modern Computing Environments. *In Proceedings of the 21st National Information Systems Security Conference,* pages 303-314, October 1998.

[Los01] Peter A. Loscocco and Stephen D. Smalley. Meeting Critical Security Objectives with Security-Enhanced Linux *In the Proceedings of the 2001 Ottawa Linux Symposium* July, 2001.

[McV96] L. McVoy and C. Staelin, lmbench: Portable Tools for Performance Analysis, *Proceedings of USENIX 1996*, January 1996, pp. 279294.

[Mil03] Mark S. Miller, Ka Ping Yee, and Jonathan Shapiro. Capability myths demolished *Technical report, Combex, Inc.,* 2003.

[Oh02] Sejong Oh and Ravi Sandhu. A model for role administration using organization structure, *Proceedings of the seventh ACM symposium on Access control models and technologies, Monterey, California, USA*, 155–162, 2002.

[Ott01] Amon Ott. The Rule Set Based Access Control (RSBAC) Linux Kernel Security Extension, *In Proceedings of the 8th International Linux Kongress,* November, 2001.

[Pre01] V. Prevelakis and D. Spinellis. Sandboxing Applications. *In Proceedings of the USENIX Technical Annual Conference, Freenix Track, pages 119–126,* June 2001.

[Pet02] David S. Peterson, Matt Bishop, and Raju Pandey. A Flexible Containment Mechanism for Executing Untrusted Code. *Department of Computer Science, University of California, Davis.* 2002.

[Pro02] Niels Provos. Improving host security with system call policies, *Technical Report 02-3, CITI,* November 2002.

[San96] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein and Charles E. Youman. Role-Based Access Control Models, *IEEE Computer 29(2): 38-47, IEEE Press,* 1996.

[Sch75] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems *Proceedings of the IEEE, 63(9):1278–1308* September, 1975.

[Sha99] Jonathan S. Shapiro and Jonathan M. Smith and David J. Farber. EROS: a fast capability system, *Proceedings of the seventeenth ACM symposium on Operating systems principles*,1999.

[Tan86] Tanenbaum, A.S., Mullender, S.J., and Renesse, R. Van. Using Sparse Capabilities in a Distributed Operating System, *Proc. Sixth Int'l Conf on Distributed Computing Systems, IEEE,* 1986

[Wri02] C. Wright, C. Cowan, J. Morris, S. Smalley, and G. Kroah-Hartman. Linux security modules: General security support for the Linux kernel. *USENIX Security Symposium*, 2002.

[Xie01] Huagang Xie and Philippe Biondi. The Linux Intrustion Detection Project *http://www.lids.org*, 2001.

[Yap05] Hemal Rathod and Roland H. C. Yap. Using Capabilities to Implement Role Based Access Control. *Submitted to the Proceedings of the seventh ACM symposium on Access control models and technologies*, Stockholm, Sweden, June 2005.

Appendix A

A.1 Creation of Master Capability using the Hook

```
static void  cbox_inode_alloc_security   (struct inode *dir, struct dentry *dentry,
int mask)
{
    struct cap_struct_master_inode *icap; /*For master capability of the inode*/
    icap = kmalloc(sizeof(struct cap_struct_master_inode), GFP_KERNEL);

    if (!icap)
            return -ENOMEM;
    memset(icap, 0, sizeof(struct cap_struct_master_inode));

    icap->inode = inode; /*point to the inode*/
    icap->i_ino = inode->i_ino; /* ObjID of the master capability */
    inode->i_security = icap;


     return 0;
}

static int  cbox_inode_post_create  (struct inode *dir,struct dentry *dentry, int
type)
 {
             dentry->d_inode->i_security->type = type; /*Type of the Object */
             return 0;
}
```

A.2 Hook implemented for `security_inode_create(dir, dentry, mode)`

```
int check_capability_file_perm (struct task_struct *task, unsigned long i_no, int file_type, cbox_file_permissions
perm)
{
        struct cap_struct_derived_file *ifile = kmalloc(sizeof(struct    cap_struct_derived_file), GFP_KERNEL);
        ifile = check_capability_list(i_no,perm); //Loads the capability in the ifile structure
        if(!ifile)
        {
          printk("You do not have the Capability to perform permission on file");
          return -1;

        }

        if((ifile->lease <= current_lease()) || (check_Negative_Capability(inode,perm) )
        {
           printk("Capability Not Valid");
           return -1;
        }

        if ((task->pid != ifile->pid) ||  (task->uid != ifile->uid)) /* Checking Stamp */
        {
           printk("Capability Forged");
           return -1;
        }

        if (ifile->type != file_type)
        {
           printk("File type is not the same");
           return -1;
        }

        return 0;

}
```

Figure A.1: Simple Logging is used here for illustration

Appendix B

B.1 Example - open.c

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main (void)
{
        int count;

        for (count = 0; count < 1000000; count ++) {
                int fd = open("hehe.txt", O_RDWR);
                close(fd);
        }
        return 0;
}
```

B.2 Example - fork.c

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

int main (void)
{
        int count;

        for (count = 0; count < 10000; count ++) {
                if (!fork())
                        return 0;
                wait(NULL);
        }
        return 0;
}
```

B.3 Example - tar.sh

```
#!/bin/bash

tar -xf linux-2.6.9.tar
tar -cf linux-2.6.9.tar linux-2.9.4
rm -rf linux-2.6.9 linux-2.6.9.tar
```