Master's Thesis in Telematics
for the Award of the Academic Degree
Diplom Ingenieur
at the
Graz University of Technology

Aspects of Access Management in Heterogeneous Distributed Object Systems

submitted by

Heimo Haub

Institute for Information Processing and Computer Supported New Media (IICM)
Graz University of Technology
hhaub@iicm.edu

October 1999

Supervisor: Univ.-Prof. Dr. Dr. hc Hermann Maurer

Assessor: Univ.Ass. DI. Dr. techn. Klaus Schmaranz

Diplomarbeit aus Telematik
zur Verleihung des akademischen Grades
Diplom Ingenieur
an der
Technischen Universität Graz

Aspekte von Zugriffskontrolle in Heterogenen Verteilten Objektsystemen

vorgelegt von

Heimo Haub

Institut für Informationsverarbeitung und Computergestützte neue Medien (IICM)
Technische Universität Graz
hhaub@iicm.edu

Oktober 1999

Begutachter: Univ.-Prof. Dr. Dr. hc Hermann Maurer

Betreuer: Univ.Ass. DI. Dr. techn. Klaus Schmaranz

Abstract

This master's thesis presents an innovative concept for user and user access management in heterogenous network systems in general and the implementation for the distributed object system Dino in particular. For this purpose the main concepts of distributed systems as well as the main concepts of Dino are introduced in order to state the requirements for a distributed user access management system. Afterwards the most common access control mechanisms and their realizations in known operating systems and distributed applications are discussed. Finally, the implemented version for Dino is introduced and its advantages and disadvantages are discussed.

An additional aspect that has influenced this thesis is the fact that Dino is meant as a basis for a medical telematic information system. This participation lead to additional requirements. As an example, the Dino system has to reach a high international standard. Therefore, the basics of the new "Common Criteria" are presented.

Kurzfassung

Die vorliegende Diplomarbeit präsentiert ein neuartiges Konzept zur Verwaltung von Benutzern und deren Zugriffen auf Ressourcen in heterogenen Netzwerken im allgemeinen und die Implementation für das verteilte Objektsystem Dino im speziellen. Dazu werden die Hauptcharakteristika verteilter Systeme und jene des Dino Systems vorgestellt, um dann die Anforderungen an eine verteilte Benutzer- und Zugriffsverwaltung aufzustellen. Anschließend werden die gebräuchlichsten Methoden und deren Realisierung in diversen Betriebssystemen bzw. verteilten Applikationen diskutiert, um daraus schlußendlich die implementierte Version für Dino herauszuarbeiten und deren Vor- und Nachteile zu betrachten.

Ein weiterer Aspekt, welcher in diese Arbeit einfloß, ist die Tatsache, daß Dino als Grundlage für ein medizinisch telematisches Informationssystem dient woraus sich zusätzliche Anforderungen ergaben, wie zum Beispiel das Erreichen eines hohen internationalen Sicherheitsstandards. Dazu werden die Grundlagen der neuen “Common Criteria” vorgestellt.

I hereby certify that the work reported in this thesis is my own and that work performed by others is appropriately cited.

Signature of the author:

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten Anderer unverändert oder mit Abänderungen entnommen wurde.

Acknowledgments

First of all my special thanks go to Klaus Schmaranz who is the spiritual father of this thesis. Not only did he give me the chance to start working on it as a member of the IICM Dino group but he also supported me in all my needs and more.

Of course everything has a beginning and so I want to thank the whole team at the IICM for their great support whenever I had demands or questions, especially to Didi Freismuth and Karl Trummer.

Additional thanks go to Karl Blümlinger and Christof Dallermassl who did not stop giving me useful information and always inspired me through a good discussion until 4 o'clock in the morning. I also want to express my gratitude to all who supported this work as readers, especially Lena Krutter and again Klaus Schmaranz.

Certainly studying and writing a master's thesis has an immense impact on ones privat life and needs additional support. So I want to thank my girlfriend Tanja who always had an ear for me and morally supported me. Last but not least also thanks to my parents who made sure that I never had to worry about anything I needed, whatever it was and also gave me moral support.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The scope of this thesis is to discuss the aspects of user access management in distributed heterogenous object systems in general and to present the implementation of one solution for the *Dino* middleware system in detail. Dino (**D**istributed **I**nteractive **N**etwork **O**bjects) is a distributed object system developed at the IICM, Graz University of Technology.

The goal of Dino is to provide an easy-to-use object system which is able to embed a broad range of network systems through a common interface. The need for a special distributed user access management as part of Dino arose through the following facts:

- Providing an integrated access management makes it possible to define easy-to-manage, global access rights. Otherwise this would have to be done separately for each embedded system.
- A main goal of the Dino design was to provide functionality on a common denominator which does not mean to reduce to a minimum (as it is done in several comparable systems). Thus, as different types of systems have different types and levels of access control (or even none), a common access management was required.
- Dino is part of the *MTP* (Medical Telematics Platform, former MTZ) project [Zwa98], whose goal is the development of a global distributed medical information system to be used in hospitals, for emergencies, etc. As medical applications always need to protect patients' data on an especially strong level, a powerful user access management was needed. MTP also stated the need for special access rights not available

in most systems such as: inheritance, case dependent user roles or the possibility to define rights for both, users and documents. As an example, doctors being in the role of emergency doctors may gain additional access rights which they normally do not have.

With these requirements in mind some additional research was undertaken which then led to the implementation of a new concept of distributed access management. The steps performed, as well as the implementation process, are described in the remaining chapters of this thesis.

1.1 Chapter Overview

Chapter 2, *Distributed Systems:* Since Dino is a distributed object system, it is necessary firstly to take a look at the basics of such systems. Chapter 2 therefore presents an overview of how a distributed system is expected to behave. Especially the design issues and the problems that may arise from them are discussed more in detail. Also the concept of the object-oriented design approach is presented.

Chapter 3, *The Dino Design:* To be able to define and understand the requirements of user access management, it is necessary to become familiar with the Dino system. This chapter presents the key concepts, such as the hierarchical address structure of Dino objects and the built-in link management. Additionally, the way how systems are embedded and how their functionality becomes available will be discussed in great detail.

Chapter 4, *Problem Specification:* Based on Dino's goals and the requirements specified by the MTP project it is possible to define the problems which a possible implementation of distributed access management is confronted with. So this chapter may be seen as a basis for the research and design work carried out for this thesis.

Chapter 5, *Access Control Mechanisms:* Before designing a distributed concept for access control it is essential to take a look at common, stand-alone system mechanisms. It will be seen that these may be reused for distributed purposes with some extensions and modifications. Therefore, this chapter also presents the basics of some common authentication and access control techniques.

Chapter 6, *Existing systems:* In order not to re-invent the wheel completely, it is of great benefit to see which solutions have already been found for similar problems. Thus, the best aspects of each system, together with additional ideas are recombined to form the desired system. In this case the best ideas of the popular operating systems Unix, Windows NT and the middleware system NetDynamics were taken as a basis.

Chapter 7, *International Standards:* As already mentioned above, Dino is intended to be the basic platform for a medical information system (MTP). For such systems, it is of high importance to achieve a high security level based on an international standard. Otherwise the system would never be accepted. Chapter 7 presents therefore such a new international standard, the so-called *Common Criteria*. Although the current implementation does not aim at a specific security evaluation level, it has always been in mind during the design process that such a level has to be reachable.

Chapter 8, *Implementation:* This chapter presents the solution found as the result of the research and development steps discussed above. It presents a highly modular and distributed way to control user access throughout the whole Dino environment without being restricted to the access control mechanism provided by embedded systems.

Chapter 9, *Outlook:* Future work to be done is summarized in this chapter.

1.2 Related Work

For a closer look at distributed object systems, especially for a comparison of Dino against other middleware systems, see [Dal99].

Chapter 2

Distributed Systems

"Alone one goes faster, together one goes further"
African adage

In order to state the requirements for distributed access management, it is first necessary to work out the main points of distributed systems. Thus this chapter will give a short overview of distributed systems with their advantages as well as some disadvantages.

We live in times of ongoing rapid changes. Particularly computer technology and its use is constantly being developed further. One of the main signs for this is the worldwide interconnection of systems in the past few years which lets us all come closer together.

Due to the hardware costs and the lack of fast connections, mainframes with some attached terminals were the standard for a long time. Two main developments changed that completely:

- cheap and powerful microprocessors and
- high speed *Local Area Networks* (LANs).

Microprocessors led to the inversion of Grosch's law, who once said: "The computing power of a CPU is proportional to the square of its price." So most companies bought the largest single machine they could afford. But from then on the reverse was true and the goal was "everyone his own computer". Of course some data exchange between the

single workstations had to take place, which led to networks. These networks connected different users on different types of systems and even at different locations. They allowed the sharing of data, CPU power and devices. This is what is basically meant by the term *Distributed System*.

Two different main kinds of distributed systems can be distinguished:

- **Homogenous** systems and
- **Heterogenous** systems.

Although both have most of their features in common, one could say that the adage "One needs to crawl before one walks" is appropriate here ([SSF97]). As homogenous networks only consist of systems with the same architecture, it is much easier to write distributed applications for them than it is for heterogenous ones.

In the starting time of networks, homogenous applications were sufficient, as most companies had only workstations of the same type they wanted to connect. But when larger networks, such as the Internet, became more and more important, the need for heterogenous applications arose. Next, it must not be forgotten that nowadays networks no longer consist only of what we would call a computer in the classic sense. Almost everything could be connected today, e.g. tomographs in hospitals or black boxes in airplanes.

Thus, the demand for applications which make distributed resources widely available and provide services for manipulating them became very high as well as the efforts in research did. For a closer look at the research done see [SSF97], [Mul89] or [MM97].

2.1 Expectations

When working in a distributed environment, users expect it to behave like one single system. They do not want to know anything about the way embedded systems or connections are handled or which kinds of systems are present at the moment. They just want to use all services provided. Therefore, there is some level of abstraction necessary which should lead to more simplicity when using the system.

As an example, users expect to log into a distributed system only once. Especially if

authentication is required by the main system (e.g. operating system) they do not see the need for providing username and password again in order to access distributed services (e.g. Unix - distributed file system via NFS). Extra logins for each embedded filesystem would be especially unacceptable because there is no visual difference between local and external filesystems.

Another point has been stated once by Leslie Lamport of DEC (Digital Equipment Corporation now part of Compaq Computer Corporation) [Lam99]:

> A distributed system is one, where I cannot get my work done, because some machine I have never heard of, has crashed.

Though this may seem extremely pessimistic, it is often true when systems are designed to rely on the existence of at least a few machines. Thus distributed systems have to hide such failures from the user.

The following section gives an overview of important issues for designing distributed systems, so that their advantages overweigh.

2.2 Design Issues

What makes a Distributed System? What are the requirements? What are its benefits and which are the important aspects for software designers? By presenting the key design issues for distributed system applications now, these questions will be discussed more in detail.

2.2.1 Transparency

One of the most important points for a distributed system is the fact how well it hides distribution. Although this may sound funny it is very important for the use of it. Besides, transparency is the main point in what a distributed system differs from a “normal” network system. There are two levels at which transparency may be implemented. In the easier case, transparency only hides distribution from the user. In the more complex and desirable case, transparency hides distribution from applications. Obviously application

transparency is much harder to achieve than user transparency as it has to provide a common and easy-to-use programming interface for applications implemented on top of the system.

One may say that transparency is the technique to make one user or one program think that it works on a single processor machine with only local resources and no other users/programs involved. This definition directly leads us to five different types of transparency known today:

Location Transparency

Location transparency is achieved when it is possible to access resources without telling where they can be found and without even needing to know where these resources are located. As an example, telling a web browser to access `http://www.iicm.edu/hhaub/myhomepage` is **not** location transparent. It should be sufficient to say: access `myhomepage`.

Migration Transparency

This means that data may be moved across the network, without changing the visible name for the user (please note that 'user' also stands for programs in this context). As an example, in Unix' NFS filesystem there is not any migration transparency. If a file is moved to a different server, it will be found under a different mountpoint (see also section 6.1.6 for more information about NFS).

Replication Transparency

If a document is very popular and thus often requested, it would be much more economic to have it residing on many different locations. In this case a request may be answered quickly and network traffic remains low. A system able to make and manage such copies on its own is said to be replication transparent. Of course there are many more things to do than just put a copy on many servers. These copies have to be kept in sync and thus replication transparency often goes hand in hand with the two following kinds of transparency:

Concurrency Transparency

As distributed systems should appear like one single system, users do not know anything about the existence of others. This leads to the possibility of two or more users trying to modify files at the same time. Anyhow, distributed systems have to use mechanisms to keep the documents' contents stable and avoid any kind of inconsistency. A simple solution to this task is the use of locks which allow only one single user at a time to access a document. Another much more complicated way is to merge the modified results in a correct manner. The method used is of course dependent from the type of application. For example, when using digital signatures, merging would be quite impossible.

Parallelism Transparency

This is the most complex part of transparency and is still one of the main fields of research. The goals are clearly defined: When writing an application for a distributed system which better runs in parallel (e.g. a chess program), programmers should not have to implement parallelism explicitely themselves. Instead, the distributed system should be able to figure out which programs would take advantage of parallelism, and how to distribute them among available processors.

2.2.2 Flexibility

The second cornerstone of distributed systems is their demand for flexibility. Single systems, often referred to as *monolithic kernel systems* provide all services through kernel functionality. Thus, adding additional features leads to a kernel recompilation or at least to a reboot of the system. Distributed systems should define common interfaces for services which can then be added at runtime. As an example supported filesystem types may change over time and should easily be added at runtime.

2.2.3 Reliability

Leslie Lamport's statement (see section 2.1) is a very pessimistic description of distributed systems. The meaning of reliability should imply the contrary and, if well designed, this is one of the big advantages distributed systems provide.

A reliable system stands on two feet, *availability* and *fault tolerance*. Availability means that if one machine crashes, there exists at least one more which can take over the job from the other one. This again implies, that data is stored redundantly. One may say, the more redundant a system is, the more reliable it is. Unfortunately it may be inconsistent as well.

Availability also implies some kind of fault tolerance. Data must not be lost through a system crash (as well as data about logged in users).

2.2.4 Performance

Performance is a measure for all distributed systems. It is in the nature of such systems to lose performance as they have to transmit messages across the network. Although parallelization is a good thing, it should be used wisely to reach good performance values. Therefore one differentiates between *fine-grained* and *coarse-grained* parallelism. Jobs belonging to the first type have many small computations to be done, whereas jobs of the second type need few large computations to be performed. Obviously, to reach the best possible performance, coarse-grained jobs are the better ones for parallelization.

2.2.5 Scalability

As we have seen in the past, one is never able to estimate the growth of specific systems. E.g. nobody thought of the Internet becoming distributed the extreme way that we can observe at the moment. So when designing a distributed application one should imagine that it will be the best system ever (of course it is) and everyone wants to use it. Scalability should be one of the main aspects of design, so that additional machines and users are easy to handle.

2.3 Advantages, Problems and their Impacts

The advantages of distributed system are clear: as microprocessors and networks are no longer a matter of cost, giving to all users their own computers is no big effort. In contrary, the benefits taken are enormous and many profitable applications developed in

recent years would otherwise not have been possible (e.g. airline reservation systems, electronic banking, ...). The following list summarizes these advantages:

- Data sharing
- Communication
- Flexibility
- Device sharing

Beside all benefits that come with distributed systems, there also arise many problems which may be classified in three categories:

- Software
- Networks
- Security

2.3.1 Software

Since software for distributed systems is quite rare, experiences with it is rare too. This leads to many different approaches in designing distributed applications and to the lack of common guidelines. But there is a light at the end of the tunnel, as distributed applications have been one of the main points of research in the last decade. See [SSF97] for a discussion about the architecture of distributed computer systems.

2.3.2 Networks

As distributed systems heavily rely on communication via networks, they suffer from all common failures accompanying them. In order to work properly, distributed applications have to be ready to expect such failures. They have to detect failures, determine the cause and react accordingly. This may lead to a reconfiguration of the system or to the initialization of a recover operation. Some problems and possible solutions are:

- **Communication-link failures:** This means messages are lost because nodes between the source and destination are not working properly. A possibility to detect this is to try sending messages a different way.
- **Node failures:** Destination nodes may crash and no longer be reachable. This may be detected by sending "are you alive" and "of course I am alive" messages.
- **Lost messages:** Messages sent may somehow be disrupted and may therefore be lost in space. Sending the same message more than once may be a good solution when a certain timeout is reached.

For algorithms that cope with such failures, see [Tan92] and [Mil92].

2.3.3 Security

Considering security, the main problem in distributed systems is that two complementary concepts collide. On the one hand it is desirable to access all resources everywhere and on the other hand access should be restricted. Thereto Andrew S. Tanenbaum says in [Tan92]:

> [...] In other words, security is often a problem. For data that must be kept secret at all costs, it is often preferable to have a dedicated, isolated personal computer that has no network connections to any other machines, and is kept in a locked room with a secure safe in which all the floppy disk are stored.

Aspects of general access control mechanisms will be the scope of chapter 5.

2.4 Distributed Object Systems

A way of increasing popularity in designing distributed systems, is the object oriented approach. The resulting **Distributed Object Systems** use the same basic technology known from object-oriented programming. This means that systems are broken down to basic objects of fine granularity. Such objects *encapsulate* both, data and behavior. Their

functionality is presented to other objects by providing *services*. Communication between objects is handled through sending *messages*.

The difference between conventional, single machine OO-programs and distributed object systems is, that in the latter case objects my reside at different locations. Thus it is one of the main design issues for such systems to hide this fact.

As we want to send messages between objects and want to access them, we need some kind of location service which keeps track of all objects in the system and which is able to route requests to the appropriate object.

NOTE: Some researchers refer to *CORBA* (Common Object Request Broker Architecture) or *RMI* (Remote Method Invocation) as distributed object systems. In fact this is not completely true, as both are rather tools to create distributed object systems than being distributed object systems themselves.

Now that the main requirements for distributed systems have been discussed it is time to take a closer look at the architecture of the distributed object system which is the subject of this thesis – Dino.

Chapter 3

The Dino Design

"... then we will do it without a network!"
Klaus Schmaranz, 5am

Dino (**D**istributed **I**nteractive **N**etwork **O**bjects) is a distributed object system which the Dino group at the IICM began to develop in 1996. The first version was based on simple message broadcast which led to many problems such as immense traffic load, weak security and the lack of a global addressing scheme. Therefore the following versions V2 and V3 used a hierarchical object addressing scheme but still suffered from some bottlenecks which led to version 4 currently under development. To get an idea about how Dino V4 is implemented, the main features and goals of Dino are described more in detail first.

3.1 Features and Goals

Dino is a middleware system providing a single point of access and is as such a virtual view to an underlying heterogeneous information space. Functionality of the underlying systems must not be lost but a superset of the functionality of all embedded systems is provided by the middleware layer to the outside world.

As a middleware layer Dino does not implement low-level database features of any kind, such as low-level indexing etc. There exist well-defined interfaces for all functionality like full-text searches, directory services, etc... that are mapped to the underlying systems by

Dino, regardless of the integrated systems.

The ultimate goal of being a middleware layer for system integration is to provide the superset of all functionality of all integrated systems. This approach also means to add features of one system to another that does not have this functionality by now. Therefore very rich standard functionality is offered by the middleware layer as a set of interfaces which are mapped to the underlying systems.

The Dino middleware has an open design allowing to embed whatever external system as long as this system provides either an API layer or a network protocol. The resulting Dino middleware main features may be summarized as follows [dTi99b]:

Platform Independence Since Dino aims platform-independence, the decision to use 100% pure Java was made.

Distributed Object System Using Dino, it is possible to access different types of network systems transparently without any loss of functionality. Thus, Dino objects may either be passive (representing data, meta-data, etc.) or active. Active objects provide additional functionality (e.g. whiteboard). Additionally, it is possible to connect two or more Dino systems to form a peer-network which then acts transparently as well.

Customization Dino provides an easy-to-use interface for customization of embedded systems. This includes providing simple mechanisms to create user specific system views (e.g. hierarchical or graph) as well as the embedding of external systems according to the desired network environment. Also the embedding of new, unknown systems is possible without recompilation or reboot.

Navigation and Views Providing customizable views to the Dino system implies that views are completely decoupled from the internal addressing scheme.

Accessibility from External Systems Dino also provides common interfaces for access from external systems which is also transparent and extendable at runtime. As an example Dino provides an HTTP gateway to be accessed by usual browsers.

Performance As already mentioned in section 2.2.4, performance is an important measure. So Dino was designed with the rule in mind not to slow down a system disturbingly compared to using it without Dino.

Scalability In order to fulfill the requirement of distributed (object) systems to be arbitrarily scalable (see section 2.2.5), Dino follows these implementation rules [dTi99b]:

> [...] the Dino kernel must not implement any algorithms with a worst-case runtime-complexity more than O(n log n) [...]
>
> [...]memory-complexity of the Dino kernel has to be configurable in a way that the system will run on computers with different memory configurations [...]

Security In addition to user access control mechanisms, which will be described more in detail in chapter 4, security features such as secure network connections between Dino systems, electronic signatures and encryption or smartcard-support are provided on a kernel level.

3.2 Realization

Dino is a modular, layer-oriented system that allows the replacement of single parts of the system without running into side-effects. Side-effects would occur if parts of one layer deal with the responsibilities of other layers.

The features discussed in the previous section are achieved through the following highly modular Dino layer model, used in Dino V4 (see figure 3.1).

The key philosophy for the Dino layer model is [dTi99a]:

> Provide all functionality that is already available to the above layers, but do never add more functionality than available by interpretation outside your scope.

Consequently each layer has its own scope and is responsible for passing on functionality already provided by the embedded system as well as to add non-existing functionality transparently to above layers. One may say that layers become more intelligent in a bottom-up manner. To understand the Dino layer philosophy let us have a closer look at the single layers:

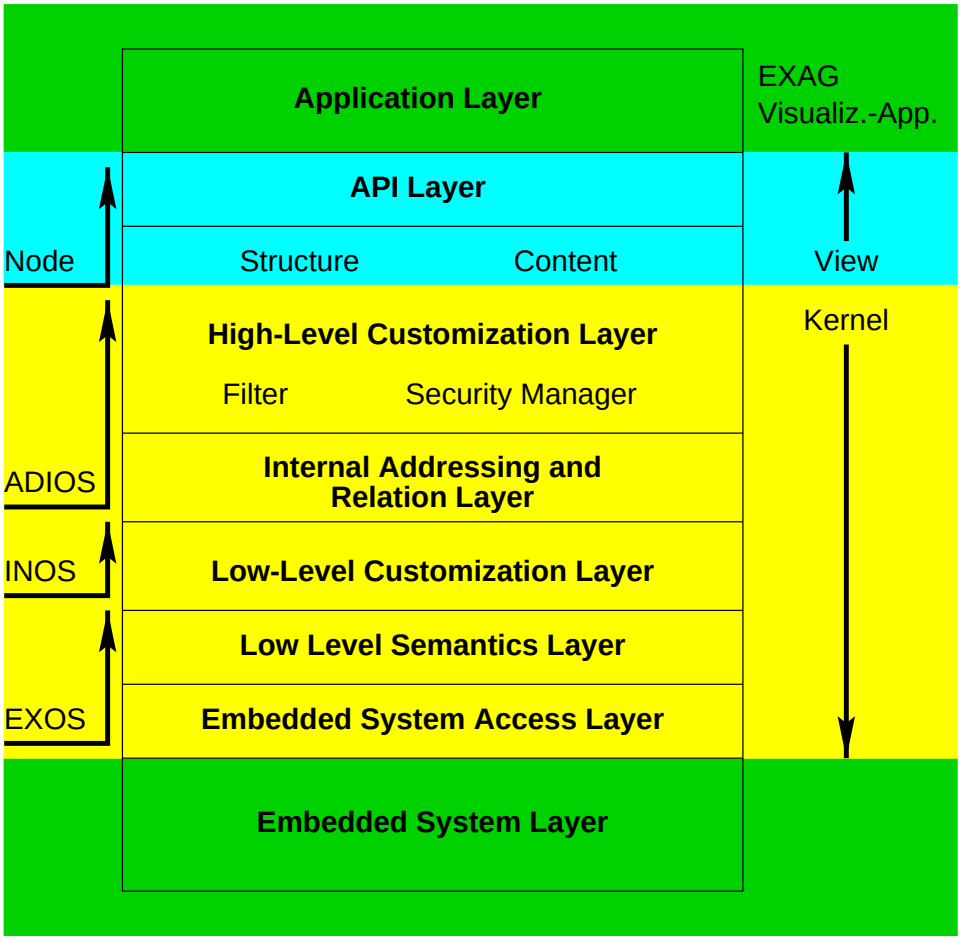

Figure 3.1: The Dino Layer Model

Application Layer Dino applications (called *Dinosaurs*) as well as external access gateways (e.g. HTTP) are residing here. Internals of the Dino system are not accessible for them. Access is solely possible through interfaces provided by the API layer. As an example the visual tool for user management is located in the application layer.

API Layer The API layer is the layer of highest abstraction. It is implemented as one single class `Dino` which provides all visible functionality such as:

- requesting a specific view
- embedding systems

- combine embedded systems to create specific configurations
- manipulate structure or relations (e.g. copy, delete, etc...)

High-Level Customization Layer For the scope of this thesis, this is the most important layer of Dino as the presented user access control mechanism resides here. This is the case because the High-Level Customization Layer separates the user visible API from the Dino kernel, whose modules are not security-checked. So access requests of the API layer have to be checked here and (if passed) forwarded to lower layers. Responsible for this is a security manager that follows customizable guidelines. This process will be described more in detail later in this thesis. Additionally, the security manager is responsible for class loading to assure that only Dino-certified modules may become part of the kernel. Other tasks accomplished in this level are:

- Dino kernel configuration management
- Dino system bootstrapping
- providing and managing different views

Internal Addressing and Relation Layer For Dinosaurs navigation in the Dino system is only possible through relations (represented through views), since the internal addressing scheme is not visible to the API layer. Therefore, the Internal Addressing and Relation Layer is responsible to map relations to internal Dino addresses.

Low-Level Customization Layer Dino provides the possibility to add additional functionality to embedded systems. This is achieved through the definition of *configuration clouds*. Such a cloud is built from several embedded systems, each responsible for handling different functionality. As an example an embedded FAT filesystem does not support attributes. Thus a configuration may contain another filesystem or database which is responsible for attributes. To the above layers this looks as if the embedded system supports attributes itself.

Low Level Semantics Layer This level maps the semantics of an embedded system to a structure conforming with Dino. As an example, database tables and entries are mapped to the hierarchical Dino addressing scheme. Relations defined in an embedded system are converted into a representation conforming with Dino as well as additional provided embedded system functionality.

Embedded System Access Layer This is the lowest layer of Dino. Low level protocols to communicate with embedded systems reside here. This is the place where external data is brought into the Dino system.

Embedded System Layer The layer containing the "outside-dino" world.

For a more detailed description of the Dino Layer model and the participating classes see [dTi99a].

3.3 Key Concepts

Dino concepts with a big impact on the problem specification and realization of a user access management system are the scope of this section.

3.3.1 Addressing Mechanism

The way objects are addressed has an immense impact on the definition of access rules and the way how to store them. Although Dino does not assume embedded systems to follow some addressing scheme, it maps all external objects into a hierarchical object tree. Additional features of external addressing schemes, such as graphs, are represented in Dino through relations (see section 3.3.2). As an example, a graph system may be mapped into the Dino addressing scheme in a flat way while all graph connections become relations.

Addressing starts with a virtual global Dino root. The following level distinguishes between local objects and those residing on other embedded Dino systems. The next level addresses configuration clouds. Below the objects themselves are addressed. For an addressing example of a Dino system composed out of three Dinos and a embedded graph and hierarchical system see figure 3.2.

Please note that embedding another Dino system is fully transparent. Thus users never get to know on which Dino system they are currently working. The addressing scheme presented above is transparent and hidden from the user.

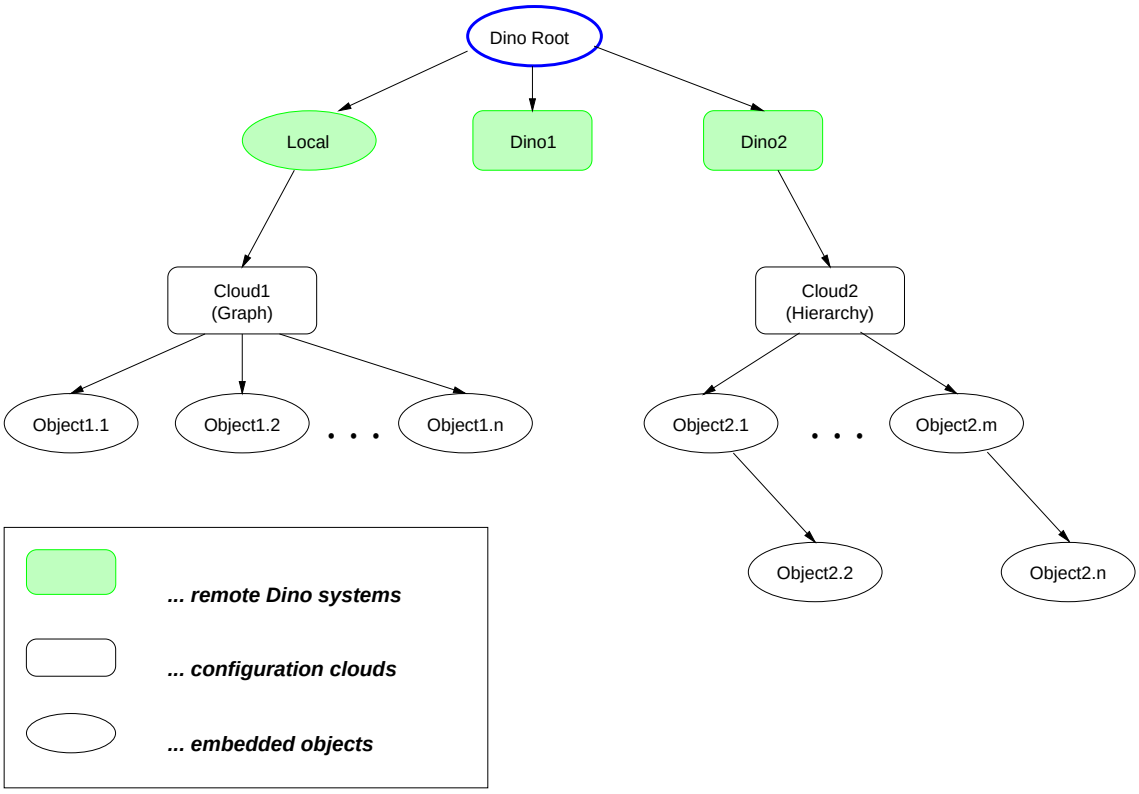

Figure 3.2: Dino Addressing Example

3.3.2 Link Management

Dino link management is aimed to overcome the following problem: Embedded systems may not provide the possibility to create links but even if they do so, mostly they do not do it across system boundaries. This means that links are mostly unidirectional (e.g. HTML links) and thus it is not possible to find out which links are pointing to a specific document.

Dino, therefore, provides a unique link management which makes it possible to define links for all documents, even if the embedded systems do not provide links. Additionally, links may be followed in both directions and, another important key point, links are held consistent even if one modifies, moves or deletes documents.

A brief summary of the link management's features gives the following list:

- add link functionality to systems which do not support links
- support links across system boundaries
- guarantee link consistency
- integrate proprietary link management of embedded systems
- add descriptive items to each link (creation date, link type, etc...)

For a detailed description of Dino link management see [Blü99].

3.3.3 Security Manager

The third key concept, from an access management point of view, is the security manager which has already been described briefly. The security manager is always involved when access is requested by a layer lying above the high-level customization layer. By replacing the security manager class, it is possible to implement other security strategies. In the implementation for this thesis the security manager loads a user manager and then forwards all access requests to it. A problem in the previous version of Dino was that the security manager was outside the kernel. Thus its requests have also been security checked themselves which then could lead to loops. Amongst others this was one of the main reasons for the Dino V4 redesign.

Chapter 4

Problem Specification

After taking a look at distributed (object) systems in general and at Dino in special, it is now time to specify the requirements for distributed user access management for Dino. These requirements are divided into two major categories, according to their origin, which will now be discussed more in detail:

1. *General requirements* arising out of the Dino philosophy.
2. *Specific requirements* arising out of the MTP project's demands.

4.1 General Requirements

As described in the previous chapter, Dino is a middleware system providing a single point of access to all resources of underlying embedded systems. As such access is handled transparently, access control mechanisms should be transparent as well. So the first requirement may be stated as follows:

> **1. Access control mechanisms shall be transparent.** One common interface to manage users and groups (create, delete, modify, etc...) as well as to define access rights shall exist. This has to be possibly independent of the way underlying systems are implementing security.

The second requirement can be seen as a result of the complexity such a distributed object system may reach by embedding numerous systems:

> **2. Extensive access rights shall be definable.** Even complex access rules should be representable. Thus it might be the best solution to provide a language for describing access rights.

A straight conclusion drawn from this requirement is:

> **3. Access rights have to be easy-to-manage.** Since access rules defined in a language may become very complex, tools have to be provided to manage such constructs easily.

Although access control and user management systems shall be available independent of the underlying systems

> **4. Access rights as well as users provided by underlying systems shall easily be affiliated.** Thus Dino access control mechanisms are not meant to be decoupled of those provided by embedded systems. In contrary, those shall be mapped into the Dino system.

As distributed systems have to be highly scalable (see section 2.2.5) there may exist an inestimable number of users. Out of this the fifth requirement arises:

> **5. Passing-on access rights to other users must be possible.** It would not be possible for system administrators to control access rights for hundreds of users. So users shall be able to delegate (inherit) some of their access rights by themselves.

4.2 Specific Requirements

As mentioned previously in this thesis, Dino is part of the MTP project. So scenarios of the MTP project shall also be realizable through the user access management system.

The first scenario deals with emergency-doctors. Obviously these have to be able to react quickly and, therefore, need more access than they might usually have. Thus

> **6. It has to be possible to define user roles.** So when in a certain role for a certain amount of time, users may gain additional access rights to perform tricky tasks.

The second scenario is that doctors need an advice from colleagues. Therefore, it might be necessary to delegate access rights which again leads to the requirement number five stated in the previous section.

As a third point:

> **7. Access rights should be applicable to users/groups as well as to documents.** As an example when granting access to a document for a specific user, the access right should be assigned to the user object. On the other hand, if granting general access to a document the access rule should be assigned to the document object. As a result, access rights for documents may be assigned to users even if the system containing the documents is not embedded.

Another important point in the MTP project is to reach a common standard. Therefore,

> **8. User Access Management must reach a high international standard** since a medical information system would otherwise not be accepted.

4.3 Strategy

Being aware of the requirements needed for a distributed access control management system, a strategy to reach this goal had to be developed. The strategy found for this master thesis will be the scope of this section.

1. Since the previous access control mechanisms were aimed to single workstation systems, the decision was made to take a look at the techniques used and to see which of them may be applied to the distributed case. An investigation of such mechanisms will be described in the next chapter (chapter 5).

2. As systems with distributed mechanisms already exist, some of those have been examined in order not to re-invent the wheel. So chapter 6 presents the techniques used in the popular operating systems Unix, Windows NT and the middleware system NetDynamics to see if those may be used as a basis.

3. To be in conformity with an international standard, chapter 7 describes the new *Common Criteria* standard. This makes it possible to draw some conclusions of how to reach a system conforming the CC.

Chapter 5

Access Control Mechanisms

"Quod licet Iovi, non licet bovi"

Ancient Roman adage

After the rise of the Internet and the first critical hacker attacks no system can live without any kind of security mechanism. Nevertheless, many software designers are thinking primarily about user-visible features of a system and tend to neglect security aspects. Thus it is extremely important to convince developers and vendors of the necessity of good security mechanisms: It is far better to prevent a violation than to catch the violator after the event.

The scope of this chapter is to give an overview of basic security concepts for local systems. Policies to protect an intranet from the outside and secure communication channels are not discussed in this document.

A very comprehensive site on Security relevant topics is [ISS99] for operating system specific reports and FAQs, as well as for papers about international standards (see chapter 7 for more details). An important influence on this chapter had [Mil92], [Tan92], [ES95] as well as [SSF97].

5.1 Authentication

Since there exists no way of controlling access to system resources without knowing who requests access, the availability and correctness of the authentication process is one of the main points of importance for a secure system.

Until today there are three different categories of methods known to "tell" the system about a user's identity:

1. Possession of a secret (e.g.: password).
2. Possession of an artifact (e.g.: smartcard).
3. Unique physiological or behavioral characteristics of the user (e.g.: retina patterns, signature dynamics).

To strengthen the authentication process combinations of these methods are used.

5.1.1 Secret Based Authentication

Possession of a secret is the most widely used and longest known way to identify users. The system stores the users' passwords and compares them to the one provided by the users at login time. Passwords have the advantage that they do not need additional hardware, but they are the least secure method, because it is relatively simple to gain or guess passwords especially if one acquires possession of the stored password file (encrypted or not). Thus when using this method choosing a good password is of high importance (see section 5.1.4).

5.1.2 Artifact Based Authentication

Possession of an artifact: Usually one possesses a magnetic card or a smartcard which is shown to the system via a card reader. In most cases this is combined with providing a pin-code. Smart cards have the additional advantage that they are also able to store the passwords as they can not be read from outside. The disadvantages of this method is the need for additional hardware that all users have to to carry around.

5.1.3 Biometrical Techniques

Unique physiological or behavioral characteristics of the user: This is the most secure way known today with the disadvantage that the required hardware is expensive.While physiological characteristics like fingerprints, retina patterns, etc. are not subject to change, behavioral characteristics like signature dynamics, voice patterns, etc. may vary depending on the user's feelings. This behavior is often interpreted as unreliability but a nervous or drunken user could be refused by such an authentication system, which might be an advantage.

5.1.4 Password Policies

As most systems use password authentication or a combination with some other techniques, choosing good passwords is a point of high importance for system security. Unfortunately, there exist just a few things systems or system administrators can do. The main responsibility here lies on the users' side by choosing good passwords. Researchers say that in most systems there exists at least one *Joe account* which is one having the same password as the account name. This is obviously an invitation for intruders. Therefore, this section is divided into two smaller parts: the first one describes what system defined password policies can do and the second one what makes up a good password.

System Policies

The best but most unpractial policy is using one-time passwords. This policy is used for very critical applications such as online banking as it would be an unacceptable effort in every day use. What can be done for user chosen-passwords is:

- Forbid Joe accounts.
- Force the users to change passwords after certain periods of time or after a certain number of logins.
- Check for some aspects of "good" passwords stated in the next section (e.g. enforce a combination of numbers and letters).

All of these methods must somehow be implemented in the systems user management. As an example in Unix this would be done by a special *passwd* program which will reject passwords not applying to the system's policy. Additionally, it has to be ensured that stored password files are not publicly accessible, even if they are encrypted.

Good Passwords

Good Passwords are situation dependent. As an example `G-58CRE` might look like a good password, but not if this is the user's license plate number. Such passwords are easy to guess and are therefore open doors. Strong passwords should contain some of the following attributes:

- lowercase and uppercase letters
- numbers
- special characters such as !, @, & ...
- be at least seven characters long
- not be user-dependent (e.g birth-date, etc)

In [GS96] the following interesting calculation is made:

> If you exclude a few of the control characters that should not be used in a password, it is still possible to create more than 43,000,000,000,000,000 unique passwords.
>
> Combining dictionaries from 10 different major languages, plus those words reversed, capitalized, with trailing digit appended, and otherwise slightly modified results in less than 5,000,000 words.
>
> From this, we can see that users who pick weak passwords are making it easy for attackers – they reduce the search space to less than .0000000012% of the possible passwords!

5.2 Access Control

Once knowing who is requesting access, it is necessary to find out if the requestor is allowed to perform the desired operation or not. Thus, from a software point of view, all objects may be seen as abstract data types allowing different operations to be applied. It is the task of the access control mechanism to ensure that only processes which are explicitly authorized perform the operations provided by these objects.

In general two types of access control can be distinguished:

- **Discretionary Access Control** is the one most often used in modern IT systems. It is usually based on a need-to-know policy which means, that someone (usually the system administrator or the owner of the file) has to know about documents and users and declares which user (or group) may access which document, mostly in a 1:1 relation.
- **Mandatory Access Control** is the other way of defining access rules. We all know this way out of some secret service movies, as it is often used in military or governmental institutions. Mandatory Access Control partitions data into classification levels and users are assigned to similar levels. Users are only allowed to access data, if their level dominates the level of the accessed data.

In the following two different solutions to the access control quest are presented.

5.2.1 Access Matrices

Defining different contexts in which processes may run, so-called *protection domains*, leads to access matrices as a form of representation. A protection domain is a collection of access rights, each of which is a pair <*object identifier, right set*>. In Unix systems, a domain corresponds to user id and group id. Domains as well as processes in domains need not be static. Access rights may change and processes may switch between different protection domains during execution. Domains may overlap which means one or more objects may be part of different domains possibly having different access rights associated with. See figure 5.1 for an example of an access matrix. A process belonging to domain D1 is allowed to take ownership of file 1, to read and execute file 2 but has no access to the CD-burner.

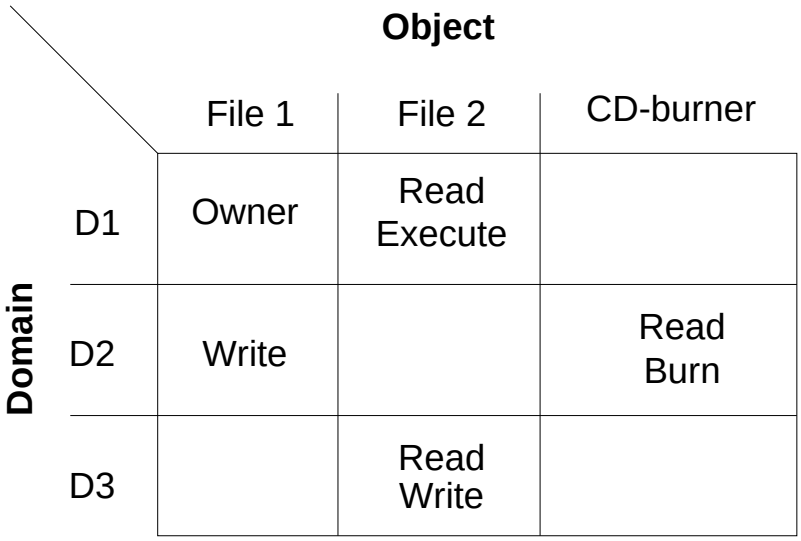

Figure 5.1: Access Matrix

Although access matrices are evident and useful they have one big drawback: they are extremely inefficient as there needs to be at least one entry for each object and thus they tend to become large and sparse.

The following sections describe other ways for representing access rights which may be expressed through access matrices as well.

5.2.2 Access Hierarchies

Access hierarchies are another way of managing different protection domains, but by inheriting access rights rather than defining them for all different domains.

Two types of access hierarchies are distinguished (see figure 5.2):

- Protection **rings** and
- Nested protection **blocks**.

Each *protection ring* defines a domain of access. The lower the number of the ring is, the more access privileges are assigned to it. For example, many systems follow the two ring

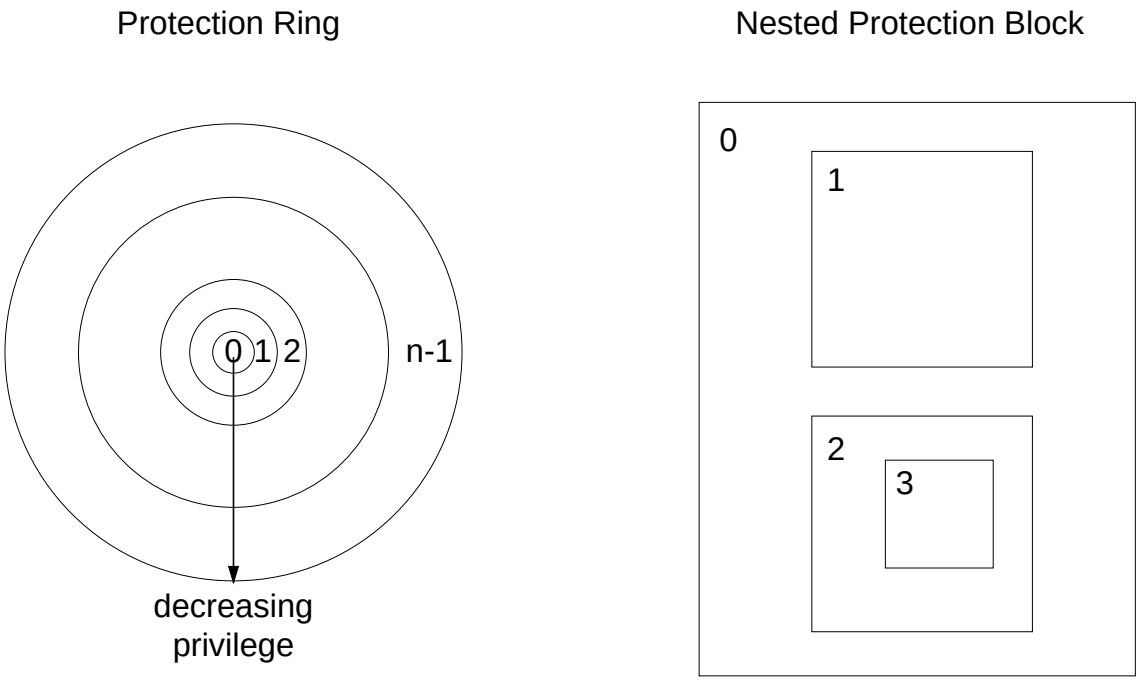

Figure 5.2: Access Hierarchies

model: Ring 0 is called kernel mode where the operating system code is executed. User programs are executed in Ring 1, named user-mode, which has fewer access privileges.

The *nested protection block* is similar to the *protection ring*. The innermost block is the one that has the most access privileges. In contrary to *protection rings* where inner rings inherit the access privileges from all outer rings, *nested protection blocks* allow a more sophisticated inheritance mechanism. As illustrated in figure 5.2, the domain 1 and 2 both inherit the access privileges from block 0, but a process running in domain 1 has different access rights than a process running in domain 2.

The disadvantage of access hierarchies is the strict ordering of access rights which makes complex constraints difficult respectively impossible.

5.2.3 Access Control Lists - ACL

Access control lists, as we will also see in chapter 6, are the most widely used form for access information storage. Usually, ACLs are defined and managed by the owner of the

object.

ACLs are built of several access control entries (ACEs), each of them defining an allow or deny rule for an operation for users and groups. One may consider an ACL consisting of all non-empty cells of a column of the equivalent access matrix (see figure 5.1). So when a process is requesting access to an object, the ACL is searched for entries of the given type for the given user. Now the system decides to either allow or deny access, following the system priorities. That means if there exists an allow rule as well as a deny rule for the requested access the system must have internal rules to solve this contradiction. Common rules may be "first come first serve" or "deny is seen stronger than allow" etc..

The main drawback of ACLs is that they should be evaluated every time a process wants to access an object, which would slow down the system extremely. As a solution to this problem some systems (like Windows NT) only check access for the requested object when it is opened. From there on the user is allowed access even if the rights change while the object is opened.

5.2.4 Capabilities

Considering access control lists may be seen as all non-empty cells of a column of an access matrix, *capabilities* may be seen as all non-empty cells of a row of an access matrix.

That means that all subjects are associated with a list of objects which they are allowed to access in some way. So processes having a specific capability get a *ticket* which allows them to access a specific object in the specified manner. So capabilities provide a single, unified mechanism to

- address both primary and secondary memory
- access both hardware and software resources
- protect objects in both primary and secondary memory

5.2.5 Conclusion

Usually, combinations of the methods mentioned above are used to control access to system resources. As an example, processes are divided to run in different modes (user and

kernel mode) according to the access hierarchy method. Additional access checks are performed using ACLs. Kernel mode processes and threads are considered trusted as they are system internal. Only processes running in user mode have to be checked additionally for authorization using ACLs.

Some systems introduce finer granularity with additional modes between kernel and user mode. As an example DEC brings a supervisor mode with more rights than the user, but less than the system mode.

5.3 Cryptography Overview

Objects may be protected by the best operating system with the best access control mechanisms, but if it is somehow possible to boot the system with a different operating system (that is, if there exist bootable devices such as floppy drives[1]) one may eventually gain access to at least all local documents.

Thus, many security experts see cryptography as the only way to intrinsically protect document access in a way that only users knowing a secret can decrypt the desired document. Nevertheless, cryptography can't prevent an attacker from deleting your data altogether. In the following a short overview about some widespread cryptographic methods will be given.

As Cryptography is a widespread subject and fills entire libraries this section will only give a short introduction to the history and some of the most important encryption methods used in modern computing. For more detailed information about cryptography and digital signatures see [Sch95], [MOV97], [GS96], [Hor96] and [Mil92].

5.3.1 History

As soon as man was able to read and write, there was the demand to write something intended for a second person that others should not be able to read. Thus, the art of cryptography can be traced back to ancient times. One of the first examples comes from Greece whose people used small stripes of parchment and wound it around a cylinder,

[1] That's why most certificates for operating systems are only valid under some configuration conditions, such as the absence of bootable devices

on which they wrote. So such a parchment stripe could only be read by a person who possessed a cylinder of exactly the same diameter like the one used by the originator.

Another example can be found in the times of Julius Caesar who used a kind of letter shifting. That means all letters in a message have been replaced by the letters that came three places later in the alphabet. As an example, using the English alphabet, the word *cryptography* would become *fubswrjudslb*.

These two methods are called *transposition* and *substitution*. Cryptography became even more important with the invention of the telegraph and the radio. The use of cryptography during World War II lead to the first computers although they where only used to decrypt messages. Later on computers also became unalterably combined with the process of encryption as they allowed thousands of complex operations to be done during a short period of time.

5.3.2 Basics

Obviously when there is a person writing secrets, there always exist persons who want to break that secret. The topic of breaking ciphers is called *cryptoanalysis* whereas *cryptology* is the combined study of cryptography and cryptoanalysis.

Encryption algorithms used in modern IT systems today may be divided into two basic types:

- **Private Key** cryptography also known as *symmetric key* cryptography uses the same key for encryption and decryption.
- **Public Key** Cryptography also known as *asymmetric key* cryptography uses a *public key* to encrypt a document and a *private key* to decrypt it.

For the topic discussed here – encryption of documents stored on a hard disk – private key cryptography is most often used. The same applies for encrypting network channels because public key algorithms are too time consuming. Public key cryptography is usually used for creating *digital signatures* as well as to exchange random session keys which are then used for a private key communication.

The most popular private key algorithms in use are:

- **DES:** Possibly the most popular one. It is also known as *DEA* (Data Encryption Algorithm) and uses a 56-bit key. To improve the security of DES it can be applied several times in a row with different keys. This method is known as *superencryption*.
- **IDEA:** The International Data Encryption Algorithm uses a 128-bit key and is used by the PGP (pretty good privacy) program to encrypt files and electronic mail.

The most popular public key algorithms in use are:

- **RSA:** It is based on the factorization problem of large primes. As it is very time consuming, it is often only used to exchange private keys for symmetric encryption.
- **Diffie-Hellman:** Similar to RSA it is based on the problem of factorization. It is used for exchanging private keys and for digital signatures.
- **DSA:** As the name says, the Digital Signature Algorithm is used for digital signatures.

Chapter 6

Existing Systems

"He that would know what shall be
must consider what hath been."
Fuller

This chapter presents three existing systems: *Unix*, *Windows NT* and *NetDynamics*. Their distributed user access management approach served as a basis for the Dino access management system presented in chapter 8.

6.1 Unix

This section will give an overview of the way Unix is keeping track of users and groups, manages their access rights and discusses how this information is used to control access to local resources as well as external ones.

6.1.1 Basics

As widely known, every user of a system is registered through an entry in the */etc/passwd* file where Unix holds all user specific information such as the encrypted password[1], the

[1]For security reasons the encrypted passwords are often stored separately in so called *shadow password files*

users' full names, the users' identification numbers (UID), the users' group identification numbers (GID) and so on. This conglomerate of user-specific information is defining a Unix account.

Similar to the */etc/passwd* file there exists the */etc/group* file which defines a relation between human readable groupnames and GID's as well as additional user-group affiliations. So users may not only belong to the main group defined in the */etc/passwd* file but can also be members of other groups. This allows more sophisticated access control by using group affiliations for special rights. It is not necessary for a group to be mentioned in the */etc/group* file to become existent. Often this is just used to define human readable names for groups.

Now the question arises how groups are recognized by the system if they are not mentioned in the */etc/group* file. Before answering this question let us have a look at one of the main points of Unix' user/group management:
User- and groupnames are just existing for convenience. Internally Unix always uses identifiers which are historically unsigned 16-bit integers (some systems also support signed and even 32-bit integers as well). These identifiers are defined through the UID and GID which have already been mentioned above.

Therefore, it is obviously sufficient for a group to be mentioned in the */etc/passwd* file to become existent via its GID (but without a name).

It is also possible to assign the same UID (GID) to more users (groups). This then leads to the fact that they are considered to be equivalent without respect to the different names. Therefore, this should be avoided for personal accounts but it could sometimes be desirable for system accounts.

6.1.2 Access Control

In this section, the way how Unix saves access information and its meaning is explained in short. The next section shows how some Unix systems use ACLs as an extension to the basic access management mechanism. The subsequent section then shows the internal processes of controlling access to resources and the treatment of special systems users (superusers).

In contrast to some simple workstation operating systems such as MS-DOS or Windows 95/98, one has to login on a Unix workstation before starting to work. After logging in users are identified for the system through their UIDs and GIDs. These identifiers are used for allowing or restricting access.

Since almost all devices or resources connected to the system are represented via file entries (normally devices are located in */dev/*) the filesystem is the appropriate place for access control. Basically there are three different types of access controllable by Unix: read, write and execute which may be arbitrarily combined to define rights either for the owner of the file, a special group or for all other users in the system. Due to the special organization of the Unix filesystem one physical file on disk may be associated to several file entries, called *hard links*. Therefore Unix creates a so-called *Inode* (index node) for each physical file containing all relevant information about the file except for its name. Directory entries simply hold a reference to such Inodes and the human readable names for them. So the Inode is the basic filesystem entry and may be seen as the basic object of Unix' file-management. Generally, Inodes contain the following information:

- The location of the item's contents on the disk.
- The item's type (e.g., file, directory, symbolic link).
- The item's size, in bytes.
- The time the file's Inode was last modified (the *ctime*).
- The time the file's contents were last modified (the *mtime*).
- The time the file was last accessed (the *atime*) for *read(), exec()*, etc.
- A reference count: the number of hardlinks (names the file has)
- The file's owner (a UID)
- The file's group (a GID)
- The file's *mode bits* (also called *file permissions* or *permission bits*).

Only the last three entries are important for user or security management since Unix provides no way to define status-dependent access rights (e.g. time dependent) as requested

Tag	Access Right	Description
r	read	Read permission.
w	write	Write permission.
x	execute	Execute permission.

Table 6.1: Types of Unix Access Rights

by the MTP project. Table 6.1 describes all possible Inode permission bits that may be set in standard Unix. Access rights are further distinguished by the owner's, group's and other's rights which may be set separately. Note that this may allow to permit execution without allowing reading, and vice versa. The execution bit allows execution of a program but also directories must be x'ed to allow making them the current directory (change to) or to open files in the directory or in any subdirectory.

File permissions also apply for devices. So one might restrict access to disks, printers, cd-burners, etc., by simply setting the correct access bits. The only things that are not affected by file permissions in the usual way are symbolic links: changing permissions for them does not lead to a change of their access bits but of the object they are pointing at. So symbolic links always seem to have all rights but those are ignored during access checking.

For more information on Inodes also see [GS96].

NOTE: In standard Unix there is no way to allow or restrict access for special users or groups. If you want to grant access to one user to a specific file this user either has to be the owner of the file or must be a member of the group associated with the file. If the user is neither the owner of the file nor a member of the specified group, access may only be granted to all other users as well. Thus fine grained access control is only possible by creating many groups.

As a solution some add-ons for Unix exist which provide a more sophisticated access control using ACLs (also see section 5.2.3), which are the scope of the next section.

ACL Entries	Description
1. attributes: SETUID	Special file modes.
2. base permissions	Standard Unix file permissions.
3. owner(owner_user): rwx	owner and access rights.
4. group(owner_group): r-x	group and access rights.
5. others: r--	other's rights.
6. extended permissions	Additional ACL entries.
7. enabled	enabled or disabled.
8. permit --x u:some_user, g:some_group	Permits access to the specified user-group combination in a boolean AND manner.
9. deny rwx g:guest_group	Forbids access to the specified user-group combination in a boolean AND manner.

Figure 6.1: Example of an AIX Access Control List

6.1.3 Access Control Lists in Unix

ACLs are not part of standard Unix but are supposed to become one in further releases. Therefore, it is not guaranteed that these access extensions also work throughout network file systems. When ACLs are used it has to be ensured that they are supported by the network file system (e.g. NFS3) as well. Although there exists no standard for ACL implementations, the basics mentioned here should apply to all known systems. Examples presented in this chapter are based on AIX Access Control Lists.

Through Access Control Lists it is possible to achieve a better refinement of the conventional Unix security system. As an example, for granting access to one specific user the system administrator would have to create a new group only containing that user and then give the file access to that group. ACLs are much easier to handle because it is possible to permit or explicitly deny access for specific users or groups.

Figure 6.1 shows an ACL as used in AIX. Lines in an ACL are called ACL entries. The first line called *attributes* defines special file modes such as *SETUID, SETGID* and *SVTX* (sticky bit). The next part called *base permissions* reflects the basic access rights that are

usually available in Unix. The second part called *extended permissions* is the ACL add-on and, therfore, the most interesting part for the moment. In this block every entry has to start with one of the following key entries:

- `specify`: Explicitely specifies the access rights for the given user-group combination (logical AND). That means, all access rights specified so far are overridden.
- `permit`: in contrast to *specify* this adds additional access rights (does not override access rights specified so long).
- `deny`: removes the specified access rights.

NOTE: If more users or groups are specified in an ACL entry, then the access right is only applied if all items are true. This means in our example, that line 8 allows *some_user* execution only if this user is a member of *some_group*.

Now the question arises what happens, if one rule allows access and another one denies it. In such a case deny is seen to be stronger and access is not granted. So in our example if *some_user* happens to be a member of the *guest_group* as well, this user will not be allowed to do anything.

6.1.4 Internals

Internally when logging into a Unix system a process associated with the user's UID and GIDs is created. When accessing a file the process' identifier information is compared to the file's Inode information of the requested type of access to decide if it is allowed or denied. Whenever a process creates a subprocess the latter inherits the same identifier information.

One user exists for whom all security checks are bypassed. This is the user with the UID 0 who is called superuser and is usually named *root*[2]. All requests of a process with UID 0 are allowed with no further access check.

A problem arises when checking access for non-root users: how should programs started gain access to files which the user is normally not allowed to access? Imagine the *passwd*

[2]The name root arises from the directory structure within a Unix system which is a tree and the user *root* is simply the one whom the root of this tree belongs.

command which changes the user's password. To accomplish this task users must be able to modify the */etc/passwd* file but if they do so, they could also change anybody else's password as well, which would lead to a catastrophe. So normal users should not be able to modify the */etc/passwd* file but the *passwd* command itself should perform this.

To fulfill this requirement Unix uses so-called SUIDs and SGIDs which stands for **s**et-UID respectively **s**et-GID. This bit set means that during execution, the program changes its UID/GID to the owner of the file instead of using the UID/GID of the user executing it. Thus the program has different access rights than the user that called it. Programs using this way of gaining other access rights are called SUID/SGID programs.

6.1.5 Managing Remote Access

Since the */etc/passwd* file is stored locally it is a problem to gain access easily to a remote workstation in the network as the user/group could not be known there.

One simple solution to this problem would be to manage one */etc/passwd* and one */etc/group* file and copy it periodically on every machine. Obviously, this method causes an immense amount of administration overhead and can lead to unstable network situations.

Again there exist some add-ons which provide a more sophisticated way of managing users throughout a network. Those solutions are based on the following two methods:

1. Use one network server to store the */etc/passwd* and */etc/group* files. Workstations then automatically ask this server for user and group informations.
2. All servers manage their own files which they send to a specified server. In return they are sent all additional users and groups defined on other severs which they add to the local users and groups.

Some of these systems are:

- Sun Microsystems' Network Information System (NIS)
- Sun Microsystems' NIS+
- Open Software Foundation's Distributed Computing Environment (DCE)

- NeXT Computer's NetInfo

See [GS96] for more information about these systems.

6.1.6 NFS

NFS is a network filesystem which makes it possible to embed file systems across the network transparently. Although NFS is also available on Macintosh, MS-DOS, Windows, OS/2, etc. this section will cover the Unix case.

NFS systems are divided into a server and a client side. NFS servers declare local directories which they make available to client machines through an entry in the */etc/exports* file. Client workstations are allowed to mount (embed) these directories into the local directory tree by specifying the remote host and the desired directory. Please note that a workstation may be server and client at the same time.

When users access a remote filesystem mounted on a client, their UIDs are forwarded to the NFS server, which performs the access check. The use of systems mentioned in the previous section (e.g. NIS) assures that these local users are also known to the NFS server. Additionally, NFS provides the possibility to map a client's UID to a different one on the server. This is often done for the UID 0 to prevent root users of one system to gain root access rights on remote machines as well.

6.2 Windows NT

6.2.1 History

When Microsoft started creating MS-DOS in the early '80ies it was intended for stand alone systems and designed without any respect to managing different users. Therfore, it was not equipped with any kind of user management. Later on networks became more popular and the need to restrict user access arose. As a consequence, more complex operating systems with a built in user management where needed.

For that reason Microsoft began the development of the network operating system Windows NT. As Unix (see section 6.1) had already proved to work well in larger intranet

systems, it was used as a pattern not only for user administration, but also for other basic system components as well as some system tools.

6.2.2 Basics

Windows NT workstations come with two standard user accounts: *Administrator* and *Guest*. Similar to Unix the *Administrator* has unrestricted rights and may not be deleted or disabled, only renamed for security reasons. The *Guest* account does not have any password and is disabled at the beginning.

All local users are stored in the *Security Accounts Manager* (SAM) database which, in contrast to Unix' */etc/passwd* file, is not human readable. Therefore, special user management tools have to be used to create or modify users. Table 6.2 [MR98] shows all attributes users may be given, where the most important ones are: *username*, *password* and *profile*. Profile specifies the location where the account's data is stored.

6.2.3 Remote Users

Similar to Unix (see section 6.1.5) users only exist locally and are not automatically available on other workstations. But accessing remote workstations requires an account and, especially on larger networks, a manual creation on any of them would be an immense effort.

To overcome this problem Windows NT defines *Domains* which specify related workstations. One of them, the *Primary Domain Controller* (PDC), stores all user accounts for a domain (*Domain Users*). Other workstations may define additional accounts (*Local Users*) but always ask the PDC for unknown users.

The arising problem of the *Domain* method is, that the PDC workstation always has to be online or all other workstations in the domain are inaccessible. This problem is solved by using *Backup Domain Controllers* (BDCs) which periodically make a read-only copy of the account information stored on the PDC. This makes the domain more fail-safe. See

[3]Normally a password should be required, but the system policy could be set in the way that users are allowed to log on without having to supply a password. Obviously this system policy is not scope of this work :-).

Attribute	Required	Description
username	yes	The name through which the user is identified to the system.
full name	no	The complete name of the user.
description	no	A brief description of that account.
password	yes/no[3]	The password.
groups	no	All associated groups for that account (Default: *All Users* and *Domain Users*).
profile	yes	The place to store the account is specified as well as scripts that are run at log on.
times	no	Specifies the period when the user is allowed to login to the system. Depending on the system policy an already logged in user may be forced to exit after this period.
login from	no	A list of all workstations from which the user is allowed to login (Default: all workstations in the domain).
account	no	Defines the amount of time the account is valid (Default: forever).
RAS	no	This may specify dial in rights for the account as well as various forms of callbacks.

Table 6.2: User Attributes

figure 6.2 for an example of an NT network using domain controllers (PDC and BDCs).

Similar to users, two different types of groups are distinguished in the case of *Domains*:

1. *Domain Groups* and
2. *Local Groups*

Domain Groups may only be created on the PDC and may only contain *Domain Users*. Since they are created on the PDC, they are valid throughout the whole domain. On the other hand, *Local Groups* may be created on all workstations in the domain but with the limitation that they are only locally valid.

By default any domain workstation creates the following group affiliations: the global

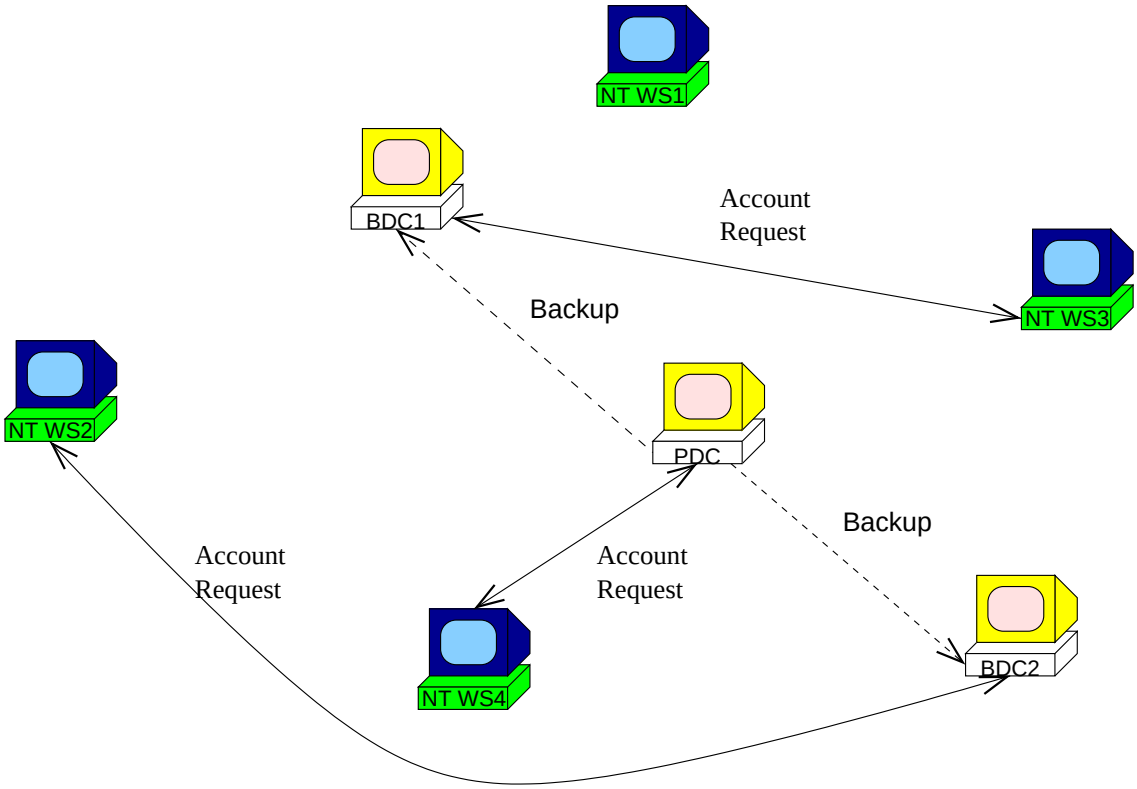

Figure 6.2: Windows NT PDC Network

Domain Users group is added to the local *All Users* group and the global *Domain Administrators* group is added to the local *Administrators* group.

6.2.4 Logon Process

It is necessary to log on to a Windows NT workstation before starting to work. Figure 6.3 shows which components are involved in the log on process. When connecting to a Windows NT system, the client is assigned a trusted process called *WinLogon* which calls the *Local Security Authority* (LSA) for identification. The LSA is comparable to the Dino *security manager* and sends the username and password to the *Security Account Manager* (SAM) for validation. The SAM also checks the account for active restrictions (e.g. disabled). If passing all checks an *Access Token* containing the user's security ID, the security IDs of all groups to which the user belongs as well as the *Everyone* group's ID and the

list of privileges that are enabled or disabled is created and returned to WinLogon. The latter then invokes the Win32 subsystem to create the initial user process with the given access token which successfully completes the logon process.

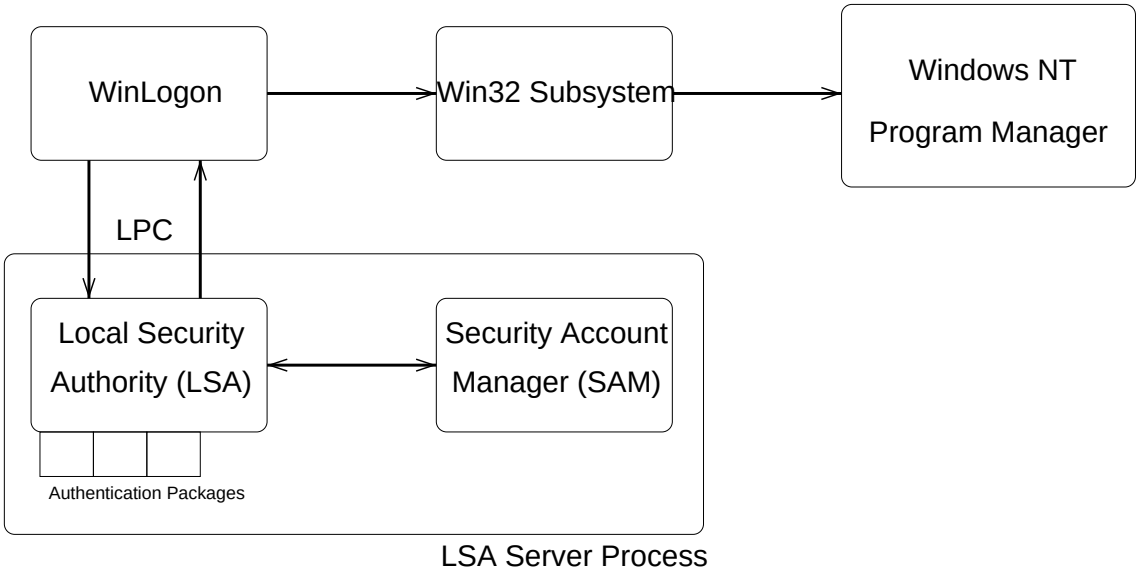

Figure 6.3: Windows NT Logon

Windows NT provides an important feature that aims distributed applications: As mentioned earlier in this thesis (section 2.1) users should not have to log in again to access distributed resources. Therefore, Windows NT offers the possibility for distributed applications to hang an *authentication package* into the LSA which is notified whenever a login occurs. These packages may then perform the authentication necessary for their applications.

6.2.5 Access Control

Similar to Unix all resources throughout the system are seen as objects and may be access-controlled more or less the same way. That means access rights are defined using *Access Control Lists* ACLs for files as well as for printers, processes, devices, drivers and so on.

NOTE: It is important to know, that file access rights can only be applied when using the *NTFS* file system. Although Windows NT also supports FAT and FAT32 for com-

patibility reasons, there is no way to define ACLs there as they are not supported. As chapter 8 describes later on, this bottleneck is overcome by Dino using a global access control mechanism applicable to all embedded systems.

All objects at creation time are assigned *security descriptors* which consist of the following attributes [Sol98]:

- **Owner SID** The owner's security ID.
- **Group SID** The security ID of the primary group for the object.
- **Discretionary access control list (DACL)** Specifies who has which access to the object.
- **System access control list (SACL)** Specifies which operations by which users should be logged in the security audit log.

An ACL (DACLs as well as SACLs) consists of several access control entries (ACEs) which differ only in their type. ACL entries of *system access control lists* define which actions on the object should be audited for which users (if successful, denied or both). ACL entries of *discretionary access control lists* are more complex: DACL entries either grant or explicitely deny access to an object. By default Windows NT grants all users full access to all resources.

NOTE: An empty SACL means that no auditing of any kind is taking place whereas an empty DACL means that no user has access to the object. If there exists no DACL at all, all users have unrestricted access rights.

The following types of access rights are applicable:

- read
- write
- execute
- delete
- permission (this permission enables users to change permissions).

- owner (allows users to take over the ownership of a file).
- full access

Windows NT applies ACLs to new objects in the following order:

1. First of all creators may specify an ACL by themselves.
2. If case 1 does not apply and the newly created object has a parent, the system recursively looks for parents that have inheritable ACEs which are then combined to form the new ACL.
3. If none of these two cases applies, the security system retrieves the default ACL of the creator's access token and applies it to the the new object.

6.2.6 Internals

According to the *mandatory access control* mechanism (see section 5.2) each process on a Windows NT system is either running in user or kernel mode. Kernel mode threads are trusted and may therefore perform all operations without being access checked. Only operations requested by processes running in user mode are security checked (*discretionary access control*).

As mentioned in section 6.2.4, the initial Win32 user process, which is created after successful login, is assigned the user specific *access token*. All processes created inherit the primary access token of the parent process and consequently all further processes in the user's session are assigned that primary access token created at login.

When processes want to access an object they have to provide a handle. Handles describe the kind of access a process (user) is allowed to perform on a specific object. The types of access for which they may be used are stored in the form of *access masks*. Using handles prevents the system from checking access whenever a process requests access. It is sufficient to check if the handle is valid for the desired type of access. Two different types of handles may be requested:

1. Handles for the maximum access allowed.

2. Handles for specified types of access. Obviously this is a subset of the maximum access allowed.

These handles are created by parsing all DACLs assigned to an object. Please note that as soon as a process possesses a handle the granted access may not be denied even if the DACL changes. This is because they are only parsed when requesting a handle. Although this may seem a bit insecure it is necessary for performance reasons.

As mentioned above, processes in kernel mode are not security checked. This is realized by using pointers instead of handles.

The following list gives a short overview of the most important components building the Windows NT Security system and is taken out of [Sol98] (also see figure 6.4)

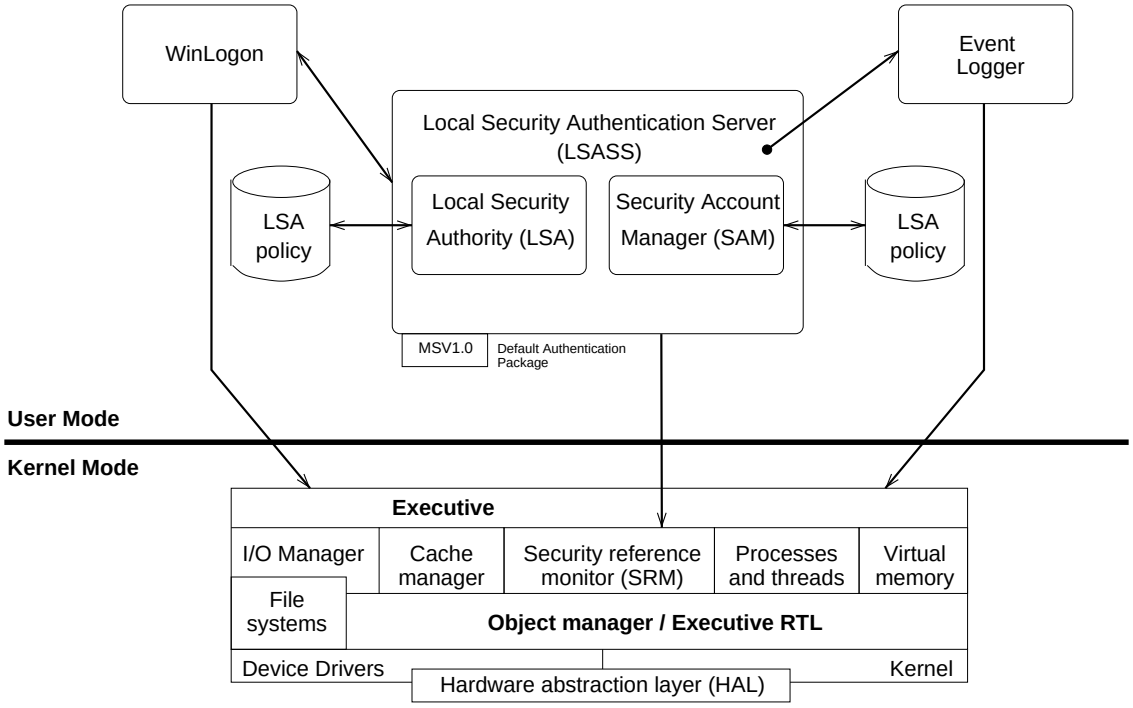

Figure 6.4: Windows NT Security Components

- **Security reference monitor (SRM)** A component in the Windows NT executive that is responsible for performing security access checks on objects, manipulating privileges (user rights), and generating any resulting security audit messages.

- **Local Security authority (LSA) server** A user-mode process running the image LSASS.EXE that is responsible for the local system security policy (such as which users are allowed to log on to the machine, password policies, the list of privileges granted to users and groups, and the security auditing settings), user authentication, and sending security audit messages to the Event Log.
- **LSA policy database** A database that contains the system security policy settings. This database is stored in the registry. It includes such information as which domains are trusted to the authenticate logon attempts, who has permission to access the system and how (interactive, network, and service log ons), who is assigned which privileges, and what kind of security auditing has to be performed.
- **Security accounts manager (SAM) server** A set of subroutines responsible for managing the database that contains the usernames and groups defined on the local machine or for a domain (if the system is a domain controller). The SAM runs in the context of the LSASS process.
- **SAM database** A database that contains the defined users and groups, along with their passwords and other attributes. This database is stored in the registry.
- **Default authentication package** A dynamic-link library (DLL) named MSV1.0.DLL that runs in the context of the LSASS process that implements Windows NT authentication. This DLL is responsible for checking whether a given username and password match to what is specified in the SAM database, and if they do so, returning the information about that user.
- **Logon process** A user-mode service that responds to network log on requests. Authentication is handled as local logons are, by sending them to the LSASS process for verification.
- **Network log on service**

6.3 NetDynamics

This section presents *NetDynamics*, which is a middleware system. Although NetDynamics does not provide such a large set of possibilities as Dino to embed other systems, its access

control management is faced with similar problems as it has to hide embedded systems' security management from other applications.

NetDynamics, distributed by Sun Microsystems, is a Web server accessible through common protocols as HTTP or IOP (used by CORBA systems). It is intended to provide a simple common access gateway to several external databases which are embedded by using general interfaces, named *Platform Adapter Components*(PACs). Existing PACs support JDBC, Microsoft COM or SAP R/3.

To see how NetDynamics has implemented a general access management for connected systems, is of advantage as it might present useful information for the implementation of the Dino access management system. These may either be good concepts or some problems that should not be replicated. For a comprehensive discussion on NetDynamics and other middleware systems see [Dal99] or the NetDynamics documentation homepage [Mic99]. An idea of the NetDynamics Application Server structure gives figure 6.5 [Dal99].

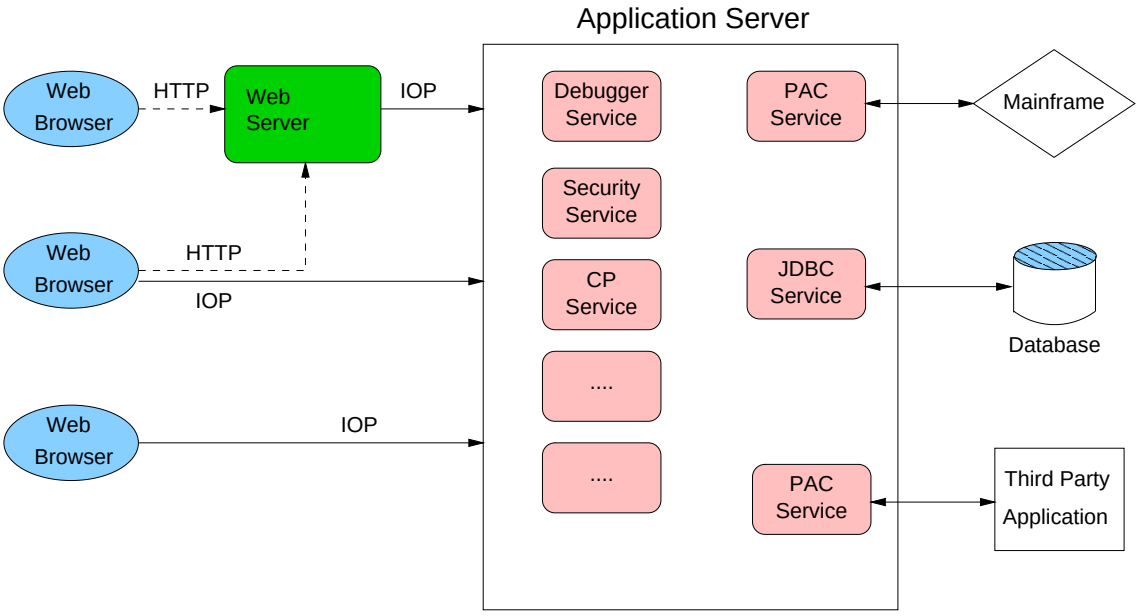

Figure 6.5: NetDynamics Application Server Structure

6.3.1 Access Control

As mentioned above, a NetDynamics server may be accessed in two different ways: by HTTP or IOP. Depending on the protocol used, NetDynamics provides two completely different views:

Web Server: HTTP clients do not gain direct access to connected external databases. Instead dynamic HTML pages are provided which are created through predefined database requests.

CORBA Application: In contrary, when using the CORBA interface applications may define any arbitrary SQL request which is sent to connected databases.

Thus NetDynamics uses two different access control mechanisms which are now described more in detail.

Web Server

When being accessed through HTTP, access checking depends on the type of connection used. That means identified control is only possible when using a stateful connection. Otherwise only access control according to system policies is possible. So NetDynamics defines two categories of access control when accessed as a Web server:

1. user-specific, if a stateful connection exists.
2. user-neutral, if the connection is stateless.

User Specific: The first user access via a stateful connection to a NetDynamics server causes the creation of a *user profile* which caches information relevant to access control. Subsequent page requests are checked against this user profile which contains the following information:

- List of pages previously visited
- Web User ID and password

- User privileges
- Database user IDs and passwords

From a security point of view the users' privileges are of interest which fall into the *mandatory access control* category (see section 5.2). This means that projects define arbitrary sets of access types (privileges) such as *read, write, execute, etc.* or *ceo, finance, admin, employee* which are assigned to documents and users. When requesting access to a specific document the users' profiles are checked whether they contain at least all attributes assigned to the document.

User Neutral: If no stateful connection exists, the NetDynamics server grants access depending on the system policy. Such policies may allow or deny access depending only on connection specific attributes such as the remote IP address.

CORBA Application

As mentioned above, applications may transmit any arbitrary SQL statement when using the CORBA interface. As these requests are not predefined, access control has to be more sophisticated than in the case of HTTP requests.

Therefore a CORBA *security manager* object is used on a service level. The security manager administrates users and groups and defines operations on CORBA objects that clients are authorized to execute through ACLs.

Chapter 7

International Standards, The Common Criteria

An important aspect for using any kind of network application, operating or middleware system is the fact how safe it is. As this is not easy to answer for complex systems, even for security specialists, several *Security Certification Authorities* (SCAs) have been founded which state some levels of security criteria. Using such criteria, customers do not have to rely solely on the assumptions developers make. Companies can have their products certified for a specific level.

Since reaching a high security level is of special importance for medical information systems it is of high interest for the Dino system (as being part of the MTP project) to reach such a high certificate.

As the *Common Criteria* (CC) seems to become more or less the most important standard, this section will present the basics of the CC evaluation processes and its levels of security.

7.1 History

The idea of security evaluation criteria was born somewhere in the '70ies and found its first standard in the US-American *Trusted Computer Systems Evaluation Criteria* (TCSEC) (also known as the "Orange Book"), which was published in 1985 by the NSA. Some

European countries took the first common effort to create an international standard which lead to the *Information Technology Security Evaluation and Certification* (ITSEC)[ITS99] criteria in 1991. The participating countries were France, the United Kingdom, Germany and the Netherlands. Other widely known standards were the *Canadian Trusted Computer Product Evaluation Criteria* (CTCPEC) and the US *Federal Criteria* (FC) which came out to replace the Orange Book. Both where published in 1993.

Since none of these standards was globally accepted, the *International Organization for Standardization* (ISO)[ISO99] began to develop one in the early '90ies. This attempt was not very successful until 1993 when the seven leading organizations of the European countries Germany, France, Great Britain and the Netherlands as well as of the United States and Canada came together to combine their different IT security criteria into a single globally accepted standard. The participating organizations, called the *CC Editorial Board* (CCEB) are:

- Communications Security Establishment (CSI), *Canada*[1]
- Service Central de la Sécurité des Systèmes d'Information (SCSSI), *France*[2]
- Bundesamt für Sicherheit in der Informationstechnik (BSI) (German Information Security Agency - GISA), *Germany*[3]
- Netherlands National Communications Security Agency (NLNCSA), *Netherlands*[4]
- Communications-Electronics Security Group (CESG), *United Kingdom*[5]
- National Institute of Standards and Technology (NIST), *United States*[6]
- National Security Agency (NSA), *United States*[7]

This coproduction led to the *Common Criteria for Information Technology Security Evaluation* (CCITSE) usually just known as the *Common Criteria* (CC). It approved as an

[1] `http://www.cse-cst.gc.ca/cse/english/cc.html`
[2] `http://www.scssi.gouv.fr`
[3] `http://www.bsi.bund.de/cc`
[4] `http://www.tno.nl/instit/fel/refs/cc.html`
[5] `http://www.cesg.gov.uk/cchtml`
[6] `http://csrc.nist.gov/cc`
[7] `http:/www.radium.ncsc.mil/tpep`

ISO standard in 1999 under the official name *Evaluation Criteria for Information Technology Security*. Please also have a look at the Common Criteria homepage [NIS99] or at any of the participating organizations stated above for further information. On these sites also the official paper [Org99] on which this section is based can be found.

7.2 Basics

The CC paper [Org99] states the following scope of the Common Criteria:

> "It is meant as a basis for evaluation of security properties of IT products and systems. The CC permits comparability between the results of independent security evaluations. It does so by providing a common set of requirements for the security of IT products and systems and for assurance measures applied to them during security evaluation. The evaluation results may help consumers to determine whether the IT product or system is secure enough for their intended application and whether the security risks implicit in its use are tolerable."

Therefore, the Common Criteria offer an easy-to-use security criteria framework for customers as well as for developers and evaluators. They make it simple to either propose or claim standards for systems they want to use (customers) or they have developed (developers).

Systems in the evaluation process are called *Target of Evaluation* (TOE), which may be operating systems or applications as well as computer networks and distributed systems.

As previously mentioned, the Common Criteria evaluation framework targets all groups involved in the IT circle (consumers, developers and evaluators) and, therefore, offers different problem definition entry points for each of them:

- **TOE Consumers** need products reaching a certain level of security. For this reason they use *Protection Profiles* (PP) to define their special security requirements. PPs are implementation independent, and consumers may either choose one out of an existing catalog or define their own.

- **TOE Developers** want to position their product on the market. Firstly, they have to find out what kind of security is required and after analyzing this they define the *Security Target* (ST) for their product, which is the equivalent to the PP but in contrary is implementation dependent.
- **TOE Evaluators:** Although the CC does not contain procedures for performing evaluations, it describes the set of general actions the evaluator has to carry out and the security functions on which to perform these actions.

PPs are guidelines for developers to see which criteria their products should meet whereas STs show costumers which security characteristics a product has.

7.3 Protection Profiles, Security Targets and Packages

Protection Profiles and Security Targets are the measures of the Common Criteria for consumer demands as well as for product claims.

PPs and STs may not be defined by good will, they also have to follow special criteria and guidelines. So before one may state or use a newly created PP or ST, it has to be certified in such a way that it is examined for consistency and completeness. Only after this, one may test a product (TOE) for compliance. This method of previously evaluating the criteria and afterwards evaluating TOEs according to the PP or ST is equal to the way it is done in the ITSEC standard.

To ease the composition of Protection Profiles and Security Targets, the CC provide a predefined catalog consisting of *classes*, *families* and *components* out of which one may build new PPs or STs. Not using these predefined objects may lead to less acceptance of the evaluation result. Of course one may also define new classes, families and components.

Classes contain families which themselves are built from at least one component. Classes are further distinguished as being of one of the two following major types:

1. **Security Functional Type:** all classes that describe functional requirements (e.g. authentication, communication).

2. **Security Assurance Type:** all classes describing assurance requirements (e.g. documentation, testing, life-cycle support).

Additionally, the Common Criteria defines packages which are "a reusable set of either functional or assurance components, combined together to satisfy a set of identified security objectives". Some special and important assurance packages defined in the Common Criteria are the seven ***Evaluation Assurance Levels*** (EALs) which represent the CC predefined assurance levels of IT products and systems. EALs levels 2-7 are generally equivalent to the assurance portions of the TCSEC C2-A1 respectively the ITSEC E1-E6 scale. For a short description of the seven EAL levels see table 7.1 which was taken from the UK ITSEC homepage[8].

Table 7.1: Common Criteria Evaluation Assurance Levels (EAL)

Level	Description
	*Inadequate Assurance.*
	*Functionally Tested.* Provides analysis of the security functions, using a functional and interface specification of the TOE to understand the security behavior. The analysis is supported by independent testing of the security functions.
	*Structurally Tested.* Analysis of the security functions using a functional and interface specification and the high-level design of the subsystems of the TOE. Independent testing of the security functions, evidence of developer "black box" testing, and evidence of a development search for obvious vulnerabilities.
	*Methodically Tested and Checked.* The analysis is supported by "grey box" testing, selective independent confirmation of the developer test results, and evidence of a developer search for obvious vulnerabilities. Development environment controls and TOE configuration management are also required.

[8] http://www.itsec.gov.uk

Table 7.1: Common Criteria Evaluation Assurance Levels (EAL) (continued)

Level	Description
	*Methodically Designed, Tested and Reviewed.* Analysis is supported by the low-level design of the modules of the TOE, and a subset of the implementation. Testing is supported by an independent search for obvious vulnerabilities. Development controls are supported by a life-cycle model, identification of tools, and automated configuration management.
	*Semiformally Designed and Tested.* Analysis induces all of the implementation. Assurance is supplemented by a formal model and a semiformal presentation of the functional specification and high level design, and a semiformal demonstration of correspondence. The search for vulnerabilities must ensure relative resistance to penetration attack. Covert channel analysis and modular design are also required.
	*Semiformally Verified Design and Tested.* Analysis is supported by a modular and layered approach to design, and a structured presentation of the implementation. The independent search for vulnerabilities must ensure high resistance to penetration attack. The search for covert channels must be systematic. Development environment and configuration management controls are further strengthened.
	*Formally Verified Design and Tested.* The formal model is supplemented by a formal presentation of the functional specification and high level design showing correspondence. Evidence of developer "white box" testing and complete independent confirmation of developer test results are required. Complexity of the design must be minimized.

NOTE that higher level EALs include all of the requirements of lower ones.

Obviously it is recommendable that, when defining a Protection Profile or a Security Target, it should include an EAL as this guarantees wide acceptance. The higher the EAL level the higher the security level and, therefore, the higher the effort of the developer to implement it and the higher the price. An overview of the composition process of PPs and STs gives figure 7.1.

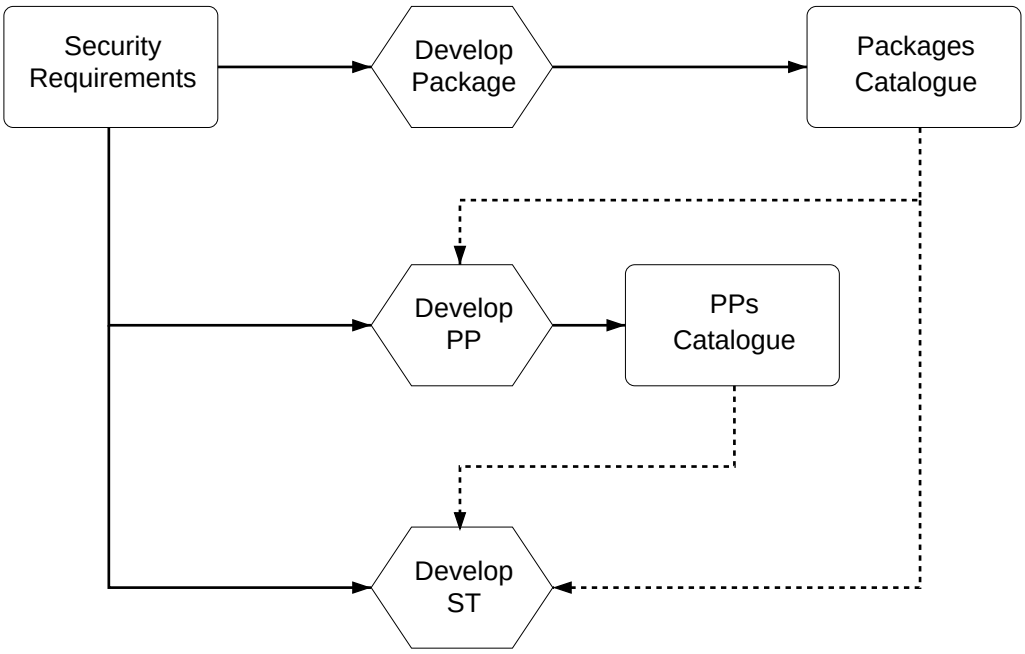

Figure 7.1: Composition Process

Protection Profiles contain the following items:

- The **Introduction** contains identification and overview information.
- A **Description of the Target of Evaluation (TOE)**. A (not necessarily security based) description of the IT product and its purpose.
- A **Description of the TOE security environment** concerning security aspects. This includes assumptions, threats and organizationally security policies.
- **Security Objectives** for the TOE and the environment.
- **IT** and **TOE Security Requirements**.

- Application Notes
- Rationale

7.4 Evaluation

Although Common Criteria do not define practical criteria for the evaluation process they state the need for an independent validation of evaluation results in order to ensure that a product's evaluation was conducted properly. Responsible for this task are *evaluation authorities*. Each of the previously mentioned organizations is such an evaluation authority. For the United States the NIST and NSA work together as one single authority and, therefore, founded the *National Information Assurance Partnership* (NIAP).

When wanting to certify a product the criteria for which it should be conducted have to be found. One of the major advantages of the cooperation of all important countries is, that the results of one authority are also accepted by the others.

An overview of some products evaluated until now is given in table A.1. The process of evaluating all components involved in the Common Criteria is illustrated in figure 7.2.

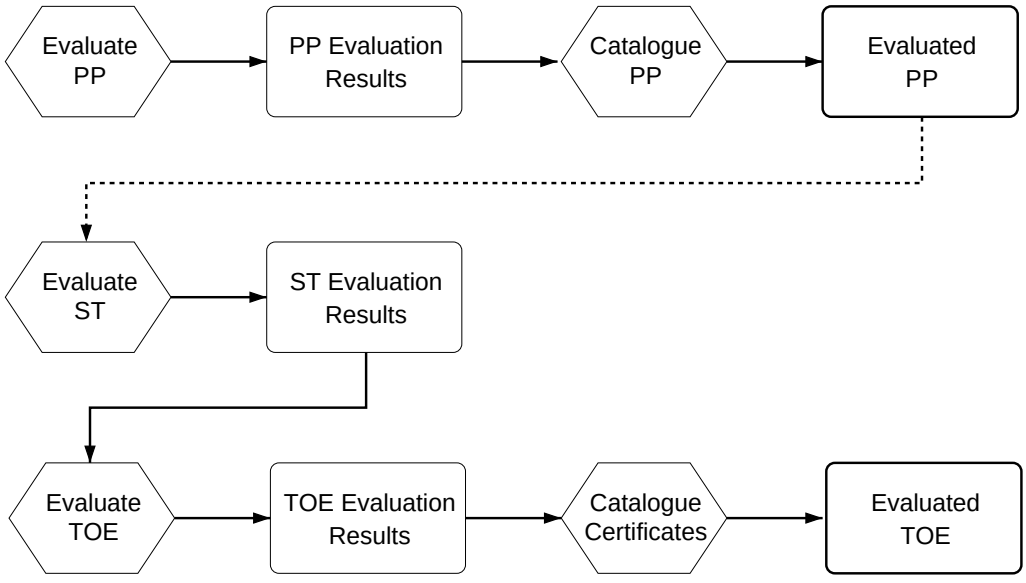

Figure 7.2: Evaluation Process

Chapter 8

Implementation

Knowing the requirements (see chapter 4) for an access control mechanism for Dino, the key techniques (presented in chapter 5) and the realizations of access control mechanisms by others (see chapter 6), the design and implementation of the Dino access management solution began. This was done with two points in mind:

1. Try to fulfill all stated requirements.
2. Invent new methods where necessary, but reuse good concepts found during the research.

First, this chapter describes the process of user authentication, whereas the remaining part presents the access control solution found, showing step by step how the requirements were fulfilled and which known concepts have been reused. Throughout this chapter the requirements stated in chapter 4 are referred to as ***R1*** to ***R8***.

8.1 Authentication

To check any kind of access the requesting user has to be identified unambiguously. As mentioned in chapter 3 the Dino security manager is the border between the kernel and the upper layers and, therefore, the best place for authentication. Most modern systems

assign to all users their own threads which identifies them throughout a session. The Dino authentication mechanism follows this approach:

Details: All users acting in a Dino system are assigned their own thread. Whenever threads request access to resources, the security manager checks if they are permitted to do so. This is verified through user specific attributes of all threads. These attributes contain all the information necessary for the security manager to check access (such as username, remote system, etc.).

The problem is to assign threads their correct attributes. Whenever users connect to the Dino system, the security manager starts a thread with a minimum set of access rights (*unidentified thread*). Afterwards users (equivalent to applications) may ask the security manager to identify their thread. A common interface exists to support all possible authentication mechanisms described in section 5.1. The underlying mechanism will be described now more in detail and is illustrated in figure 8.1:

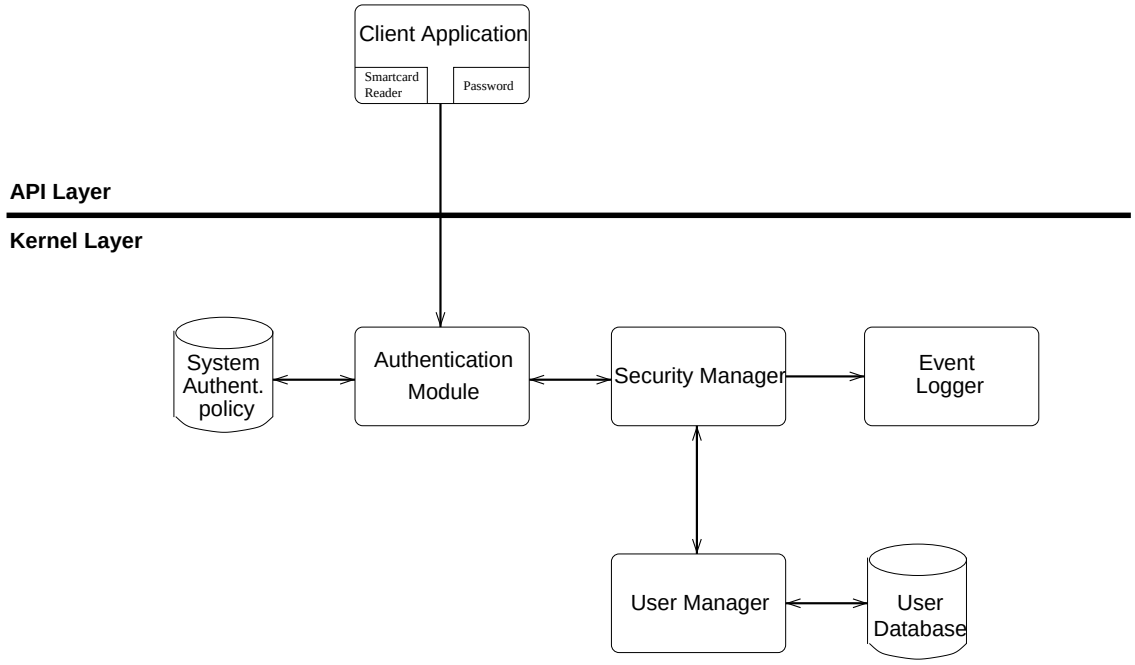

Figure 8.1: Dino Authentication

When requesting authentication or re-identification the client application submits all supported authentication types to the security manager and may additionally mark one as the

preferred one. This is done because different clients may support different mechanisms. Some client workstations may have smartcard-readers or retina-scanners connected, others may only provide password authentication. The security manager chooses one of these or rejects the authentication request if no ("strong" enough) method, according to the system policy, is supported by the client. Depending on the chosen method, a token is filled out and forwarded to the user manager. The current user manager version only supports password authentication and thus this token contains *username*, *password* and the *remote server*. The user manager then compares this information to the information stored in its repository.

If the token information passes the check the security manager allows identification and assigns the correct attributes to the thread, which are:

- *username*
- *remote server*

Whenever this thread is requesting access to resources then, these attributes are used for access control which will be discussed more in detail in the next section.

8.2 Access Control Basics

Dino is able to embed external systems transparently in such a way that there is no difference between local and remote objects. To reach this level of abstraction access control has to be transparent as well. That means when identified to Dino, no additional identification or action is necessary by users to access any kind of resource (***R1***). Thus, the decision was made to place the user access management on top of the embedded systems and not to map access information to the security mechanisms provided by embedded systems. As an example, when embedding a Unix filesystem all access rights would have to be mapped to those available (see section 6.1.2), which may be quite impossible in some cases. This point is of special importance since embedded systems may not even provide any access control mechanisms at all. According to the Dino philosophy, the Dino user management mechanism has to provide this additional functionality which is only possible

when assuming that even no kind of access control may be supported by embedded systems at all.

Another point is that a common interface for administrators as well as for users (applications) has to be provided to allow easy creation and modification of access rights, users and groups. Otherwise the access control tool would not be generally usable but depend on the embedded systems.

Detail: The Dino user access management system, named *user manager*, is divided into two major modules which have absolutely nothing in common and are separately implemented.

1. One module responsible for *identification* and access request *checking*.
2. One module responsible for *creating* and *managing* access rights, users and groups.

The major difference is that since the identification module has to check access requests it must have full access to the whole system without any restrictions or further checks (otherwise it would have to check its own requests). Therefore, it is implemented as a part of the system kernel (*High-Level Customization Layer*, such as the *security manager*). For security reasons this module does not provide any methods or services available for applications through the Dino API.

In contrast, the second module is used to manage domains (e.g. home directories) through creating and modifying access rights, users or groups by administrators and normal users. Obviously, these actions have to be checked and, therefore, this part resides above the kernel in the API layer (see chapter 3). It is clear that if there are no access rights defined it is not possible to create any rights at all. This hen-egg problem is solved by creating a standard administrator account with full access when setting up a new Dino system.

The basic implementations of both modules forming the user manager are described now more in detail:

Access Checking Module: Access requests of client applications are intercepted by the security manager. Bypassing the security manager to achieve direct access to lower layers from the API layer is impossible. The security manager creates a token which holds the desired types of access (read, write, execute, etc...) and the thread's user

attributes. This token is sent to the user manager which parses all responsible access rights to determine if the request is allowed or denied. As an alternative, the security manager may follow some system policies like: "all users of the server *remote1* have read access on all documents." In the latter case, only all others than the read requests are forwarded to be checked by the user manager.

Access Manager Module: Since access management tools have to be accessible to normal users through the Dino API, this module is implemented as a simple Dino object. Thus it is accessible for users and applications the same way as all other Dino resources. Administrative tools are provided through methods and services. Methods may be seen as interfaces for other applications whereas services provide a graphical interface (for information about methods and services in general see [dTi99a]). So the access manager module's methods and services represent the interface to clients for creating and managing access rights.

Using methods and services leads to high modularity and extendibility since they are loaded from a factory at runtime and thus are exchangeable without recompilation. This fact also follows the Dino philosophy of all system parts being highly configurable and exchangeable. Provided services by the user manager are described more in detail later in this chapter.

Access rights, users and groups are also represented by Dino objects. This has the advantage that the user manager uses the common Dino interface to access and store data. By this the storage locations may be distributed across the whole Dino system transparently.

Storing access rights separately from the resources they are applied to, brings up the problem of how to represent their relation. A simple solution possible would be to write the resource's address into the access right's data. But this method has the disadvantage of loosing these relations if the resources' addresses change (e.g. when being moved). Additionally, it is impracticable when checking access to an object, since all access rights have to be searched for the appropriate address. If all access rights are sorted by the resources' addresses, this can be performed in *O(log n)* time

The solution to this problem is one of the key-techniques of this implementation: Relations between access rights and resources are represented by using the Dino *link management* (see section 3.3.2). As a result, these relations are always held

consistent by the Dino system and are traceable in both directions. For the same reasons user-group affiliations are represented by links as well. Please note that access rights may be assigned to users as well as to documents. This is illustrated in figure 8.2.

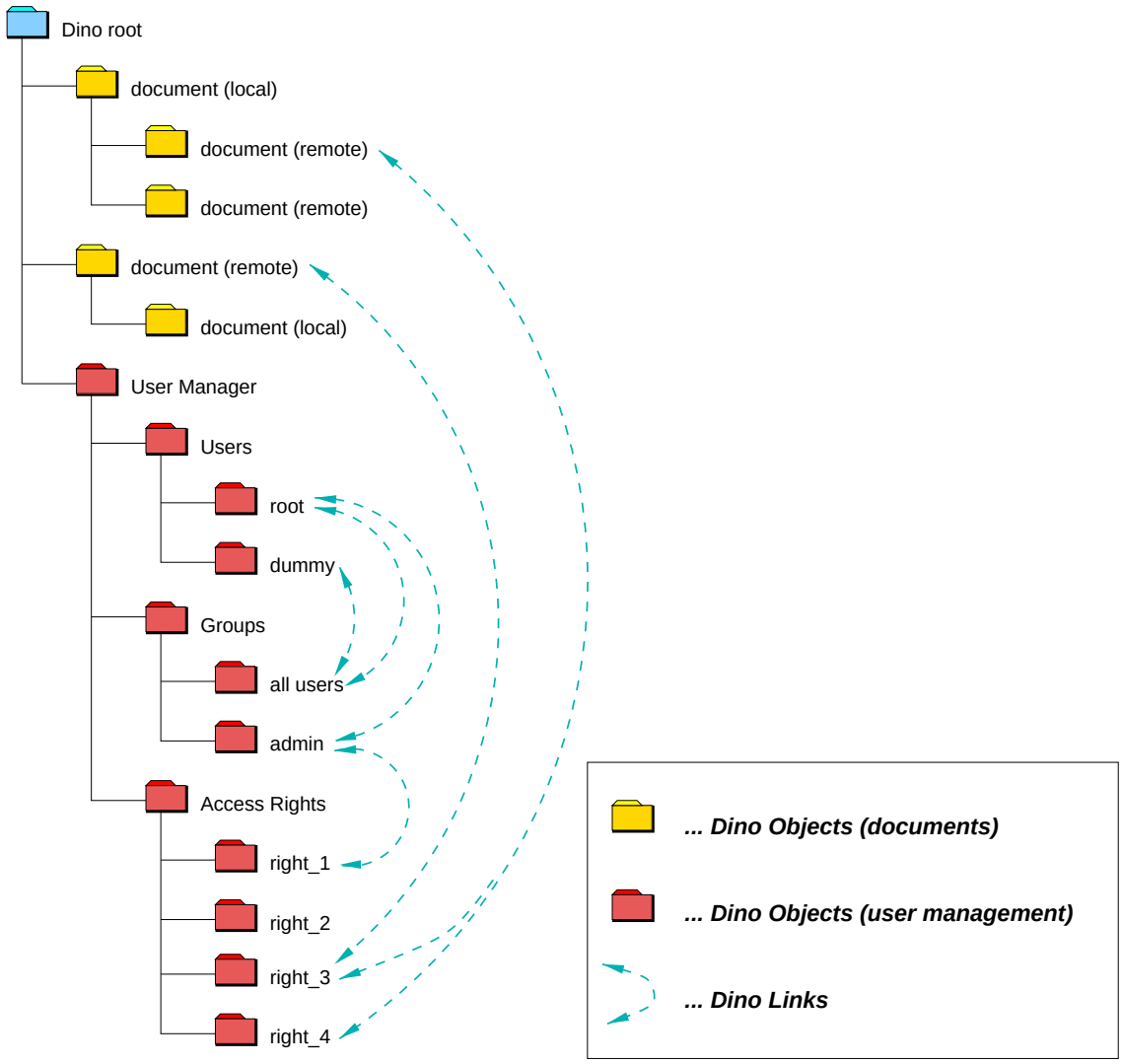

Figure 8.2: Dino User Manager Example

This figure shows a Dino environment configuration consisting of local and remote documents and a user manager. Additionally, it demonstrates how users, groups, documents and rights might be combined through links to grant access as well as to create user-group affiliations. As an example, user `root` is a member of the groups `all_users` and `admin` and therefore inherits access right `right_1` assigned to the `admin` group. If user `root` wishes to access `document_1` the following access rights are investigated: `right_3` and `right_4` (as they are assigned to the document) and `right_1`.

Please note that one right may be applied to more than one Dino object. Thus same rights do not have to be defined redundantly. How access rights are composed and how they are parsed will be described throughout the remaining part of this chapter.

8.3 Access Control Features

Describing additional remarkable features of the Dino access control management is the scope of the following section. Additionally, the way how the requirements ***R2*** to ***R7*** are fulfilled is described more in detail.

8.3.1 Extensive Rights

Requirement ***R7*** states that access rights have to be assignable to users and groups as well as to documents. Using links provides a convenient way to solve this problem. When checking access, the user manager has to look for access-right-links of the requested document as well as of the user object and parse all of the right objects they are pointing to.

Referring to section 5.2, this method is a combination of access control lists (ACLs) and capabilities. Users and groups may be seen as the *domains* (rows of the access matrix), documents as the *objects* (columns of the access matrix). This method is another key-technique of Dino access control since most systems only allow defining access rights either for documents or for users, but not for both. It is also in the mind of the Dino philosophy to provide additional functionality for embedded systems.

Standard usage of ACLs and capabilities as seen in chapter 6 (Unix, Windows NT, NetDynamics) only provides simple access rules such as "allow user x 'read-access' to document y". In our case such simple rules are not sufficient to fulfill requirement ***R2*** as stated by the MTP project, as access rights should be case dependent. As an example, doctors may want to delegate access to a patients file to nurses only for a short amount of time (e.g. to enter a blood picture). Such an access right would look like "allow write access to nurse x from 10 am to 1 pm".

Detail: The next key technique of Dino access control is the definition and usage of an access rule language. As it is out of the scope of this thesis to present the complete language semantics, only a brief introduction is given.

Two different types of access rules are distinguished:

- Allow
- Deny

If an an *allow rule* evaluates to true, access is permitted whereas it is denied in the case of a *deny rule*. Both may be nested *boolean expressions* where operands are of the two following types:

1. `attribute` *`operator`* `'constant'`
2. `attribute` *`operator`* `attribute`

`attribute` may be any possible attribute of a Dino object involved in the access check process (user, group and document). Examples for such attributes are: *username* or *profession* in the case of a user object or *creation date*, *owner* in the case of documents. Please note that attributes are not predefined. A `constant` may be any comparable value.

An overview of all implemented *`operators`* is given in table 8.1. Since operators are loaded through a factory new operators can be added at runtime.

An example of an access rule is illustrated in figure 8.3. At the beginning the version of the language used to describe the rule is specified. The section *Rules* then defines either *deny* or *allow* rules. In the given example, lines 3-5 deny write access if requested by `user1` or

Operator	Symbol	Description
EQ	==	equal
LT	<	less than
GT	>	greater than
LE	≤	less equal
GE	≥	greater equal
IN	∈	subset of
contains		does it contain the pattern
startsWith		is the pattern a prefix
endsWith		is the pattern a postfix
NOT	¬	negation, may be prefixed to all other operators

Table 8.1: Operators of the Dino Access Rule Language

`user3` or if it is requested between 1am and 4am (maybe because at that time the backup is done). Lines 6-7 allow read and write access to `user2` but only if it is a doctor or an OP-nurse. Read and write is also permitted if the requesting user is the author of the document.

As can be seen such rights might become very complex and so the evaluation process might become very slow. Consequently, these rights should be used wisely and only nested if absolutely necessary. On the other hand such a powerful language might not be needed for some Dino systems. A simple one would satisfy the needs (e.g. such ACLs as Windows NT provides). This is the reason why the rule version number is also given (see line 1 of our example). Depending on the version number a factory loads the correct evaluation object which again makes the whole mechanism exchangeable. Of course, more than one version may be used in parallel.

8.3.2 Easy to Manage

As previously mentioned such access rules might become very complex and keeping track of which user has which kind of access is very difficult. Thus to fulfill requirement ***R3*** a

```
1.[Version] 2
2.[Rules]
3.   DENY(write) if
4.      (username == 'user1') OR (username == 'user3')
5.      OR ((sys-time GT '1am') AND (sys-time LT '4am'))

6.   ALLOW(read, write) if
7.      ((username == 'user2') AND (profession IN ('doctor', 'OP-nurse')))
8.      OR (username == author-name)
```

Figure 8.3: A Dino Access Rule Example

graphical user management interface is provided (as a user manager service). It provides an easy-to-use interface for administrators to overlook and manage existing rights as well as to create new ones. Access rules may be created in either of the two ways:

1. By simply composing graphical items.
2. By using an editor to directly enter rules in the form shown in figure 8.3.

The graphical interface may also be used by users to manage their domains. This includes creating new access rights for documents they own as well as passing-on access rights (as described in the next section).

8.3.3 Passing-on Access Rights

The possibility to pass-on access rights to others, as stated in requirement ***R5*** is realized through adding a special right attribute: *pass-on*. Only rights with that attribute may be passed-on and only if one has the *pass-on* right one may set the *pass-on* attribute for other users.

8.3.4 Roles

A key feature claimed by the MTP project is the possibility of assigning roles. As an example, doctors being temporarily in the role of emergency doctors have to be granted different and additional access rights respectively.

In the current version of the Dino access control management this is realized by normal user-group affiliations and by using another important feature of the Dino link management: The possibility to assign attributes to links [Blü99]. These attributes are used to define conditions on which the user manager decides if users are currently a member of a group or not.

For the definition of a role a group is created (e.g. emergency doctor) and the correct access rights which users in this role should have are assigned. Users being members of such groups are not permanent members. But if the condition (link attribute) evaluates to true they are and thus inherit all defined access rights. Such a role may be defined as illustrated in figure 8.4 where `user_x` acts in the `role_y` every Monday from 6am to 4pm and therefore gains all access rights defined in `right_role_y` during this period.

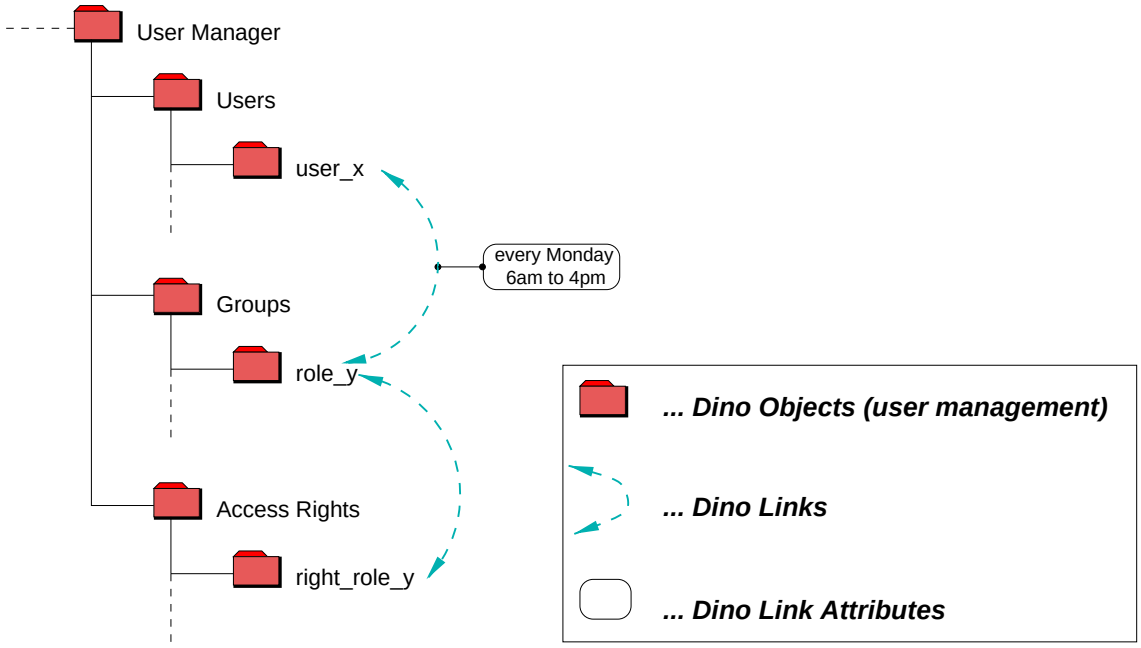

Figure 8.4: A Dino Role Example

8.4 Evaluation

The scope of this section is to describe which actions are performed between requesting access and the decision of either granting or forbidding it (Throughout this section the access rights used conform to the previously explained Dino access rights language).

As explained earlier in this chapter, users and applications are visible to the security manager through the threads they are running in, which may either be anonymous or identified. Depending on the system policy two ways exist to handle anonymous access:

1. The security manager decides on a built in system strategy which objects of the Dino system anonymous users might access without asking the user manager.
2. A guest user may be created and anonymous threads simply run in this context. Thus the access control is performed the same way as for identified threads.

If identified threads request access to a Dino object, the security manager builds a token representing the owner of the thread. Such tokens consist of *username* and *remote server*. This token in combination with the address of the requested Dino object as well as the requested type of access is forwarded to the user manager which performs the following sequential steps:

1. Fetches the user object from the user database.
2. The user object's attributes are checked. (as an example the user might be temporarily disabled)
3. All user-group affiliations are checked and a list of the groups the user is currently a member of, is created. Please note that this procedure is done recursively for all groups as groups might themselves be members of other groups.
4. All access right links of the user object as well as of all groups are traced to create a list of all access rights.
5. The same is done for the requested document object with a single addition: Since access rights may be defined to be recursively valid, the Dino tree is watched upwards for such recursive rights which are also added to the list.

6. All access rights found are checked: Each of these verifications begins with the extraction of the access right's version number and loading the corresponding *evaluation object* through a factory. Please note that if either the extraction of the version number or the loading of the *evaluation object* fails, the process is stopped and access is denied. When checking access two problems occur which are solved through the definition of system policies:

   (a) First of all, what happens if no allow or deny rule is found corresponding to the request? Does the system allow or deny access? As an example, Windows NT grants all access to all users if there is no ACL found for the object.

   (b) The second question is, what happens if one rule allows and another denies access?

   In both cases a default value has to be defined. The implemented version always prefers "deny" as this is more restrictive and thus more secure.

   Based on these policies all access rights are checked and a decision is made which is returned to the security manager.

7. Depending on the user manager's result, the security manager allows or denies access.

8.5 Merging Dino Systems

In the Dino language the term *merging* stands for the combination of two or more Dino systems. Following the Dino philosophy this is done transparently as well. This means in the case of addressing that all objects are accessible through the same address independent of the Dino environment working on. These addresses are named *Globally Unique Dino Handles* (GUDHs) and are described more in detail in [dTi99a]. For this work it is sufficient to know that they are unique throughout the whole system and held consistent against movement and modification.

Using GUDHs guarantees that objects, access rules are pointing to, are always the same and do not depend on the Dino environment configuration. The arising problem which has to be solved by the access management when Dino systems are merged is, how access

is checked for users that request access to Dino objects residing on a remote system. Therefore, from a Dino system's point of view, two different types of mergeable Dino systems exist:

1. *Trusted* Dino systems and
2. *Untrusted* respectively unknown Dino systems.

Both of them may be merged and external users may access local Dino objects, but with a major difference: Trusted systems' users are mapped to locally known ones and may, therefore, gain explicit access rights. On the contrary, users of untrusted systems are completely unknown and may only access objects that are released for public access (method of least privilege).

Figure 8.5 illustrates the process of merging two Dino systems. Part a) shows both systems before being merged. When the merging process is initiated both systems search their *Trusted Systems* table. In our example "Dino 1" trusts "Dino 2" but not vice versa. Therefore, "Dino 1" creates a user mapping table for users of the "Dino 2" system as defined in its system policy, whereas the mapping table of "Dino 2" is empty.

Whenever a user of "Dino 2" requests then access to an object residing on "Dino 1" this request is forwarded and access is checked for the corresponding user of the merged system. As an example `user_A` gains access to all objects `user_3` has access to. In reverse all users of "Dino 1" will be seen as anonymous users on "Dino 2" and, therefore, only gain access to public objects.

This method of merging two Dino systems provides a very secure way of controlling access to a system as the local system policy defines all trusted servers and which access rights remote users gain. Please note that it is also possible to define user mapping rules such as: "All remote users being member of the remote group X gain all access rights that local users being member of group A have".

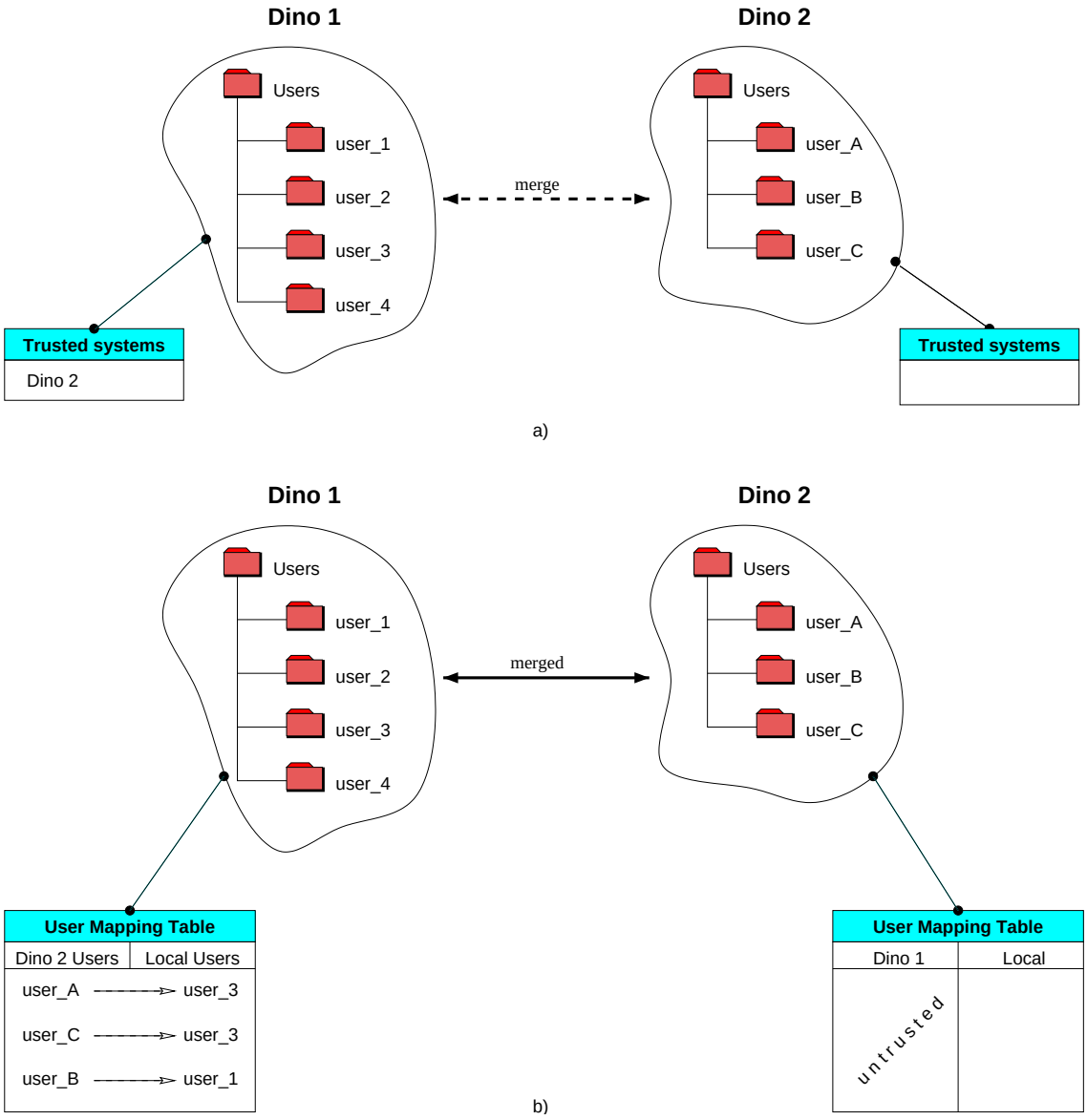

Figure 8.5: Example of Merging two Dino systems

Chapter 9

Summary and Outlook

The aim of this chapter is to summarize the key features of the implemented Dino access control management system. Additionally, this chapter presents some concepts for improvements which should be the scope of future work.

One of the main problems in making a distributed system transparent is to hide the security mechanisms of connected systems from applications. Thus, the decision was made to place the solution for the Dino access management system on top of the embedded systems. Not only does this fact cause transparency but it also makes the system highly modular. The main problem encountered during the research done was, that most of the available systems do not provide a common access management for heterogenous network systems. Most systems are only able to manage distributed access for homogenous systems. As an example, the Unix' add-ons as NIS (see section 6.1.5) or the Windows NT PDC method (see section 6.2.3) only work with equal systems. NetDynamics is the exception as it provides a common interface for connected systems. It prevents transparency, though as the access control mechanisms that are used depend on the way applications are connected to a NetDynamics server (see section 6.3.1).

The problem with the implemented Dino access management system so far is that access control mechanisms of embedded systems are not integrated by now. External systems are embedded with superuser rights and access restrictions for users and applications are only defined on a Dino system level. As a result, it will be one of the main points of future work to define a common interface which makes it possible to import access rights of embedded

systems into the Dino access management system. This will also be necessary in order to achieve an international standard as required by the MTP project (e.g. Common Criteria, see chapter 7).

Although the access control mechanisms used by the Dino access management system, namely *Access Control Lists* and *Capabilities*, already existed, the concept of using them in this extensive combination is new and unique, as other systems only use either of them. This makes it possible to assign access rights to users as well as to documents (just to that object where they are easier to manage). Additionally, commonly used ACLs are only built of simple constructs (as in Windows NT), thus providing a powerful access right language may be seen as another key feature of the Dino access management system.

A reason for other distributed systems not to provide a common access control mechanism might have been the problem of assigning access rights to external documents and resources, without assuming that any kind of access control is supported by embedded systems at all. The outstanding Dino feature used to overcome this problem is the Dino link management which provides the possibility to create links to external documents even if the systems they are residing on are not embedded. In addition, these links are consistent against movement and modification.

Since each access to resources has to be checked, another important measure for an access control mechanism is the amount of time an access check needs. The Dino mechanism implemented evaluates all responsible access rights each time which obviously leads to delays. There exist some ways to reach a better performance, which should be the scope of further discussion:

1. First of all access rights could be cached. That means, the access rights' data or an evaluated representation (maybe in a binary form) could be held in memory which would make access control much faster.

2. Another way may be the usage of tickets, similar to Windows NT. This means when wanting to gain access to a Dino object, users first request a ticket for specified types of access. If the user is allowed to perform this actions, a ticket is returned which may then be used for any access of the requested types without any further checking.

Although both methods induce an immense speed-up they suffer from the disadvantage

that it is very difficult or even impossible to keep track of any access right changes. As a result, users might access resources although the assigned access rights could have changed to deny access. Thus, any method implemented to speed-up access checking has to be chosen wisely.

From a scalability point of view the process of merging two or more Dino systems as presented in this thesis may also be refined. At the moment two Dino systems that should be merged need an extensive knowledge of the other system. Although this is obviously the most secure way for performing a merge, especially for the MTP project, other Dino environments could request a simpler way. Therefore, other possibilities should be implemented. As an example, not only other Dino systems could be marked trusted but also their user manager. Thus, access to remote Dino systems could be checked by local access rights (e.g. `user_x` is allowed to access all documents concerning heart attacks).

As a result one can say that the user access management system for Dino presented in this thesis is following a new concept by combining existing mechanisms already in use (such as ACLs and Capabilities) with some outstanding features which the Dino system provides (such as the link management). The result is a highly modular, extendable and customizable access management system for distributed object systems.

Appendix A

IT Security Certificates

A.1 Common Criteria

Table A.1 gives an overview of all Common Criteria certified products till September 1999. It shows the evaluated EAL level as well as the certification authority country and the date of certification. Whenever possible, it also displays a link to the certification report for further information.

Table A.1: Common Criteria Certificates

Supplier Name ***Product Title***	**Level-Date**	**Country, Report Link**
Bull S.A. and IBM Informationsysteme Deutschland GmbH *B1/EST-X Version 2.0.1 with AIX Version 4.3.1*	EAL 4 March 1999	Germany, `http://www.bsi.bund.de/aufgaben/ii/zert/index.htm`
Oracle Corporation *Oracle7 Release 7.2.2.4.13*	EAL 4 Sept. 1998	United Kingdom, `http://www.itsec.gov.uk/cgi-bin/cplview.pl?docno=106`

Table A.1: Common Criteria Certificates (continued)

Supplier Name ***Product Title***	**Level-Date**	**Country, Report Link**
The Knowledge Group *VCS Firewall Version 3.0*	EAL 1 March 1999	United Kingdom, http://www.itsec.gov.uk/cgi-bin/cplview.pl?docno=116
MIS Europe Ltd *SeNTry 2020*	EAL 1 July 1998	United Kingdom, http://www.itsec.gov.uk/cgi-bin/cplview.pl?docno=103
Entrust Technologies Limited *Entrust/Admin & Entrust/Authority from Entrust/PKI 4.0a*	EAL 3 March 1999	United Kingdom, http://www.itsec.gov.uk/cgi-bin/cplview.pl?docno=130
Cisco Systems, Inc. *PIX Firewall 520, Version 4.3(1)*	EAL 2 Dec. 1998	United States, http://www.radium.ncsc.mil/tpep/epl/CCentries/TTAP-CC-0002.html
ITT Industries *Dragonfly Guard Model G1.2, Software Release 3.0*	EAL 2 Oct. 1998	United States, http://www.radium.ncsc.mil/tpep/epl/CCentries/TTAP-CC-0001.html
Lucent Technologies *Lucent Managed Firewall Version 3.0 (Build 150)*	EAL 2 Jan. 1999	United States, http://www.radium.ncsc.mil/tpep/epl/CCentries/TTAP-CC-0003.html

Table A.1: Common Criteria Certificates (continued)

Supplier Name ***Product Title***	**Level-Date**	**Country, Report Link**
Voltaire Advanced Data Security *2in1 PC(TM) Version 1.21*	 June 1998	United States, `http://www.radium.ncsc.mil/tpep/epl/CCentries/TTAP-CC-0004.html`
MILKYWAY Networks Corporation *Black Hole Firewall version 3.01 E2 for SPARCstations*	 Aug. 1997	Canada, `http://www.cse-cst.gc.ca/cse/criteria/english/mbh.htm`
Signal9 Solutions *ConSeal Private Desktop for Win95/98*	 May 1999	Canada, `http://www.cse-cst.gc.ca/cse/criteria/english/s9cpd.htm`
Entrust Technologies Limited *Entrust TrueDelete version 4.0 for Win95/NT*	 May 1999	Canada, `http://www.cse-cst.gc.ca/cse/criteria/english/etd.htm`

A.2 ITSEC

As there exist hundreds of ITSEC certified products, table A.2 gives only an overview of the most important respectively wideliest known products, certified till September 1999. It shows the evaluated E level as well as the certification authority country and the date of certification. Whenever possible, it also displays a link to the certification report for further information.

NOTE that E1-E6 is comparable to the Common Criteria EAL2-EAL7 levels.

Table A.2: ITSEC Certificates

Supplier Name ***Product Title***	**Level-Date**	**Country, Report Link**
IBM Corporation *PR/SM for IBM S/390 CMOS Computer Sy-stems Family 9672-G5*	E4 March 1999	Germany,
IBM Informationssy-steme Deutschland GmbH *AIX V 4.3*	E3 May 1998	Germany,
Setec Oy *Setcad 202 Software Version 1.4.3*	E4 Feb. 1998	Germany,
Utimaco Safeware AG *SafeGuard Sign& Crypt SDK Version 2.0*	E2 May 1999	Germany,
Oracle Corporation *Oracle7 Release 7.3.4.0.0*	E3 Dec. 1998	United Kingdom,
Informix Software Ltd *INFORMIX Online Dynamic Server V7.23*	E2 March 1998	United Kingdom,
Motorola Ltd *Motorola Smartcard Chip*	E3 April 1997	United Kingdom,

Table A.2: ITSEC Certificates (continued)

Supplier Name ***Product Title***	**Level-Date**	**Country, Report Link**
Network Associates International Ltd *Gauntlet Firewall v3.01 for Windows NT*	E3 June 1999	United Kingdom,
Microsoft Ltd *Microsoft Windows NT Workstation and Server Version 4.0*	E3 March 1999	United Kingdom,
Microsoft Ltd *Microsoft Windows NT Workstation 3.51*	E3 Oct. 1996	United Kingdom,
Sun Microsystems, Inc. *Sun Solaris 2.6*	E3 Jan. 1999	United Kingdom,
SCO *SCO UnixWare Version 2.1.0*	E2 Feb. 1999	United Kingdom,
Hewlett Packard Ltd *HP-UX Version 10.20*	E3 Feb. 1999	United Kingdom,
Bull Information Systems Ltd *BEST-X/C2 (Bull Enhanced Security Technology) 1.1.1.9*	E3 June 1999	United Kingdom,

Table A.2: ITSEC Certificates (continued)

Supplier Name ***Product Title***	**Level-Date**	**Country, Report Link**
Digital Equipment Corporation *Digital UNIX Version 4.0b*	E4 Jan. 1999	United Kingdom,
IBM UK Ltd *IBM PR/SM ES/9000*	E4 Sept. 1995	United Kingdom,
Motorola Semiconducteurs *Bull CP8 Composant MC68HC05SC0401 masqué pour l' application SCOT300 (référence ZC438408)*	E3 June 1998	France,
SGS-Thomson Microelectronics SA et Bull CP8 *Composant ST16SF44A masqué pour l' application SCOT400 Version 1 (référence ST16SF44ARHQ)*	E3 April 1998	France,

Bibliography

[Blü99] Karl Blümlinger. Dino link management. Technical report, IICM, Graz University of Technology, 1999.

[Dal99] Christof Dallermassl. Aspects of integration of heterogenous server systems in intranets - the java approach. Master's thesis, University of Technology, Graz, October 1999.

[dTi99a] Dino Team IICM. Dino V4 software engineering document. Technical report, IICM, Graz University of Technology, 1999.

[dTi99b] Dino Team IICM. Dino V4 software requirements document. Technical report, IICM, Graz University of Technology, 1999.

[ES95] Jan H.P. Eloff and Sebastiaan H. Solms, editors. *Information Security - the Next Decade*. Chapman & Hall, 1995.

[GS96] Simson Garfinkel and Gene Spafford. *Practical Unix & Internet Security*. Computer Security. O'Reilly & Associates, Inc., second edition, April 1996.

[Hor96] Patrick Horster, editor. *Digitale Signaturen*. VIEWEG, 1996.

[ISO99] ISO. International organization for standardization (iso) homepage, 1999. `www.iso.ch`.

[ISS99] ISS. Iss - internet security systems, 1999. `http://www.iss.net`.

[ITS99] ITSEC. The uk itsec scheme, 1999. `http://www.itsec.gov.uk`.

[Lam99] Leslie Lamport. Home page, 1999. `http://www.research.digital.com/SRC/personal/Leslie_Lamport/home.html`.

[Mic99] Sun Microsystems. NetDynamics documentation series, 1999. `http://www.netdynamics.com/support/docs/`.

[Mil92] Milan Milenkovic. *Operating Systems, Concepts and Design.* McGraw-Hill International Editions, second edition, 1992.

[MM97] Christoph Maier and Luis Mandel. Object-oriented developement of distributed systems - a survey. Technical report, Ludwig-Maximilians-University Munich, December 1997.

[MOV97] Alfred J. Menezes, Paul C. Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, 1997.

[MR98] Ashley J. Meggitt and Timothy D. Ritchey. *Windows NT, Benutzer Administration.* O'Reilly & Associates,Inc., german edition, 1998.

[Mul89] Sape Mullender, editor. *Distributed Systems.* Addison-Wesley, 1989.

[NIS99] NIST. Common criteria project - homepage, 1999. `csrc.nist.gov/cc`.

[Org99] Common Criteria Sponsoring Organisations. Common criteria for information technology security evaluation, August 1999. Version 2.1.

[Sch95] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C.* John Wiley & Sons, 1995.

[Sol98] David A. Solomon. *Inside Windows NT.* Microsoft Press, second edition, 1998.

[SSF97] Richard L. Shuey, David L. Spooner, and Ophir Frieder. *The Architecture of Distributed Computer Systems.* Addison-Wesley, 1997.

[Tan92] Andrew S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[Zwa98] Bernhard Zwantschko. The MTZ project. Technical report, IICM, Graz University of Technology, 1998.

Index

D