

System and security model of GIDIS Trusted Linux 0.1.1  
Z version

Maximiliano Cristiá  
`mcristia@fceia.unr.edu.ar`

Grupo de Investigación y Desarrollo en Ingeniería de Software  
Facultad de Ciencias Exactas, Ingeniería y Agrimensura  
Universidad Nacional de Rosario, Argentina

December 20, 2003

## **Abstract**

This document describes the system model for GIDIS Trusted Linux 0.1 (GTL 0.1), and the security model to be verified by GTL 0.1. The system model is a formalization of an enhancement of the standard Linux Virtual File System interface that includes multi-level secure (MLS) controls and ACLs. Both models are based on the notion of information flow rather than the Bell-LaPadula security model (which is the case of Lisex, GTL's predecessor).

The system model is divided in three parts: a description of the environment state, a description of many operations that may affect security (they include a number of file system calls), and some abstract data types (ADT) that describe low level data structures.

# Contents

<b>1</b>	<b>Overview of the Model</b>	<b>5</b>
1.1	Factors that Affect the Usability of MLS Systems . . . . .	5
1.1.1	Moving the Access Control . . . . .	5
1.1.2	All Inputs Are Not Equally Important . . . . .	5
1.1.3	Access Classes of New Objects . . . . .	6
1.1.4	Empty Objects . . . . .	6
1.2	Guiding Principles . . . . .	6
1.3	Design and Implementation Comments . . . . .	7
1.3.1	Key Security Requirements of the GTL Formal Model . . . . .	7
1.3.2	Functional and Security Requirements . . . . .	8
1.4	Style conventions . . . . .	9
<b>2</b>	<b>The State of the File System</b>	<b>12</b>
2.1	Trusted Computer Base . . . . .	13
2.2	Physical Terminals . . . . .	13
2.3	Logical Terminals . . . . .	15
2.3.1	Design and Implementation Comments Regarding Terminals . . . . .	17
2.4	Users Allowed to Work on the System . . . . .	18
2.5	Protected Objects . . . . .	19
2.5.1	The Access Class of Directories . . . . .	20
2.6	Processes . . . . .	20
2.7	The State of the Environment . . . . .	21
<b>3</b>	<b>Operations Controlled by the User</b>	<b>23</b>
3.1	Chinsec . . . . .	24
3.1.1	Design and Implementation Comments . . . . .	26
3.2	Input . . . . .	26
3.3	Login . . . . .	27
3.3.1	Design and Implementation Comments . . . . .	29
3.4	The Interface for the User . . . . .	30
<b>4</b>	<b>Operations Controlled by Processes</b>	<b>31</b>
4.1	Chobjsc . . . . .	31
4.2	Chsubsc . . . . .	33
4.3	Close . . . . .	34
4.4	Create . . . . .	36
4.4.1	Design and implementation... . . . .	38
4.5	Exec . . . . .	38
4.5.1	Design and Implementation Comments . . . . .	40
4.6	Fork . . . . .	40

4.6.1	Design and Implementation Comments . . . . .	41
4.7	Link . . . . .	41
4.8	LinkS . . . . .	42
4.9	Mmap . . . . .	44
4.10	Open . . . . .	47
4.11	Oscstat . . . . .	48
4.12	Read . . . . .	49
4.12.1	Design and Implementation Comments . . . . .	51
4.13	ReadLT . . . . .	51
4.14	Rename . . . . .	52
4.15	Setuid . . . . .	54
4.16	Stat . . . . .	57
4.17	Write . . . . .	58
4.17.1	Design and Implementation Comments . . . . .	60
4.18	WriteLT . . . . .	61
4.19	The Interface to be Used by Processes . . . . .	63
4.20	Other operations . . . . .	63
<b>5</b>	<b>Operations Controlled by the System</b>	<b>65</b>
5.1	Get . . . . .	65
5.2	Put . . . . .	67
5.3	System Internal Operations . . . . .	68
<b>6</b>	<b>The Transition Relation</b>	<b>69</b>
<b>7</b>	<b>Formal Security Model</b>	<b>70</b>
7.1	Invariants of <i>UsersAndTerminals</i> . . . . .	70
7.2	Invariants of <i>LogicalTerminals</i> . . . . .	71
7.3	Invariants of <i>Users</i> . . . . .	71
7.4	Invariants of <i>FileSystemObjects</i> . . . . .	71
7.5	Invariants of <i>Process</i> . . . . .	72
7.6	Secure File System Properties . . . . .	72
7.7	The Missed Property . . . . .	73
7.8	Simple Security . . . . .	74
7.9	Where Can Users Work? . . . . .	75
<b>8</b>	<b>Subject Security Classes</b>	<b>77</b>
8.1	Basic Types, Parameters, and State Definition . . . . .	77
8.2	Operations . . . . .	78
8.3	Proof Obligations . . . . .	81
<b>9</b>	<b>Security Classes</b>	<b>82</b>
9.1	Basic Types, Parameters, and State Definition . . . . .	82
9.2	Operations . . . . .	84
9.3	Proof Obligations . . . . .	86
<b>10</b>	<b>Access Control Lists</b>	<b>87</b>
10.1	Basic Types, Parameters, and State Definition . . . . .	87
10.2	Operations . . . . .	88
10.3	Proof Obligations . . . . .	90

<b>11 Types, Parameters, and Schemas Used to Specify Operations</b>	<b>91</b>
11.1 Basic Types . . . . .	91
11.1.1 Error Reports . . . . .	91
11.1.2 Basic Modes . . . . .	91
11.2 Global Parameters . . . . .	92
11.3 Schemas Used to Specify Operations . . . . .	93
11.3.1 DAC Preconditions . . . . .	93
11.3.2 Opening an object at the <i>Process</i> level . . . . .	94
11.3.3 Common Errors . . . . .	95
<b>12 Main</b>	<b>97</b>

# List of Tables

2.1	Control of variables of <i>PhysicalTerminal</i> and <i>UserAndTerminals</i> . . . . .	15
2.2	Control of variables of <i>LogicalTerminals</i> . . . . .	16
2.3	Control of variables of <i>Users</i> . . . . .	19
2.4	Control of variables of <i>FileSystemObjects</i> . . . . .	20
2.5	Control of variables of <i>Process</i> and <i>ProcessList</i> . . . . .	22

# Chapter 1

## Overview of the Model

In this chapter we explain our motivations, goals, and principles in writing GLT's security model. Also we comment on how to map this model to an actual implementation on the Linux kernel.

Our main goal is to develop a secure UNIX-like operating system. Our second goal is to get an usable implementation of it. By UNIX-like we mean an operating system with the “same” interface than some free or proprietary version of UNIX, we choose Linux. In our vocabulary, secure means resistant to Trojan horse attacks against confidentiality [9, 1]. Finally, for us, usable means that ordinary users perceive the necessary stronger security only when it is really needed; more precisely, we would like an operating system in which security does not affect users who obey the rules [9].

### 1.1 Factors that Affect the Usability of MLS Systems

The literature clearly shows that the only way to have an operating system resistant to Trojan horse attacks against confidentiality, is to implement a multi-level security (MLS) model. Given the experience we gained by developing and using Lisex (GTL's predecessor), we know that BLP-like models [2, 3] are secure but severely reduce the usability of the system. We have identified some key factors that produce this second, undesired side effect. We will comment on them in the following sections.

#### 1.1.1 Moving the Access Control

To exercise MLS access controls at **open** time is perhaps the most influential factor. It is not clear whether a process violating confinement [3] will indeed violate security. Only when this process tries to downgrade information by writing it to a lower level file, security is about to be compromised. Thus, this time we followed an information flow model [7].

We have applied a simplification of Denning's model to a subset of Linux's system calls. This subset includes all file system calls, and calls regarding the creation and modification of processes. Also *pipes* have been considered. This is a simpler version than Denning's because we considered only explicit information flows. For example, the information flow that results from a process deleting all the possible file names from a given directory and deducing the erased names from the value returned by `unlink`, has not been considered. Also, as Denning did, we do not consider information flows through covert channels.

#### 1.1.2 All Inputs Are Not Equally Important

Other important factor that we believe reduces the usability of the system is not taking into consideration that users do not always work with classified information. In other words, in a modern computing environment, computers are used to process sensitive and non sensitive information. In order to eliminate this factor we decided to take an input-output view of the system rather than

an strictly state approach. In this view, input is classified at different access classes accordingly to user desire; and classified output is sent by the system only to appropriate terminals. This approach allowed us to represent a user entering input classified at many different access classes, and seeing output as classified as the terminal where he is working on.

### 1.1.3 Access Classes of New Objects

Yet another important factor that we have identified as contributing to decrease system's usability, regards the initial access class assigned to recently created objects. In Lisex we followed a rather obvious approach: to assign the access class of its creator when a new object is created. While this policy is indeed secure it also severely affects the usability of the system because users are committed to classify all their information at their own levels, thus contradicting what was stated in the previous section.

In the present model, new objects have the lower bound ( $L$ ) on the set of security classes. The justification is simple: new objects contain no information thus it is unnecessary to classify them above  $L$ .

### 1.1.4 Empty Objects

The last paragraph of the previous section gave us further insight on the significance of empty objects and how they should be managed. Given that an empty object does not contain information, it is impossible to disclose information by arbitrarily, and even discretionary, changing its access class. Hence, the model presented in this document was adapted to treat empty objects as fundamentally different from non empty objects. Clearly, this decision will make an implementation a little more complex because it has to deal with one more case. However, we believe that this change will increase the usability of the system. Consider the following two examples.

- A user creates a new file or directory, the system assigns  $L$  to it, and then the user has the chance to set its access class to the most appropriate.
- A pipe between two processes is defined, the system assigns  $L$  to it, and then, given that the pipe is empty, the system assigns the access class of the information that is first written on it. Moreover, given that a pipe is emptied when the other process reads enough from it, chances are to transmit information at another access class through the same pipe.

One can argue that the first scenario allows an attacker to trick the user in believing that the access class of the new object is the one he wanted, when in fact a Trojan horse has set a different (lower) one. Although this is true, to be successful this attack needs a negligent user, because the information that will be written in this new object comes either from:

- a. User input, in which case the user must classify it accordingly.
- b. Other object, in which case the system will prevent a process to downgrade it.

Negligent users render the most secure system a castle with cotton walls.

## 1.2 Guiding Principles

Besides the general principles of computer security [9], we have based the construction of this model on the following ones:



- `root` is an ordinary user with respect to MLS.

Most of the software used in a UNIX-like operating system was installed by users with access to the `root` account. It is incorrect to assume that these administrators are trustworthy as the most sensitive information managed by the system. Hence, they must be trusted as much as their access classes, and so every process acting on behalf of any of them cannot be trusted more than its owners.

- Things must start at  $L$ .

New, empty objects must be classified at  $L$ ; the first process of a user must be started at  $L$ ; the input of the user should be initially classified at  $L$ ; directories should not be classified above  $L$  unless file names are significative, and so on. MLS systems tend to increase the classification of information, basically because the impossibility of *writes-down* [11]. Thus, it is convinient to mitigate this tendency by krafted the system in a manner that it put energy to keep information at a low classification (without compromising security, of course). We think that a good design principle to follow is that the system should start things as low as possible.

- The problem is that users cannot see information they are not authorized to see.

The problem is not that users cannot modify information, nor that processes cannot read information, or even write it. If the system prevents users of seeing information they are not authorized to see, then the system is secure. It does not matter what the system do with information, nor what the system permits processes to do. Users can see information only when it leaves the system: users cannot read a file, they can only read from a screen or a printed sheet of paper. Hence the system must be designed by putting hard controls around its borders, and not necessarily inside it.

## 1.3 Design and Implementation Comments

### 1.3.1 Key Security Requirements of the GTL Formal Model

The factors that in our opinion reduce the level of usability of the system and the principles described in the previous section, leaded us to specify the following key security requirements<sup>1</sup>:

- Terminals have assigned two possible distinct access classes. One of them, *mptsc*, applies to the output sent by the system to the terminal, and the other, *cptsc*, applies to the input entered by the user (a flesh and bones human being) at the terminal. *cptsc* can be set by the user to inform the system on how high is the input that will be entered from that time on. In turn, *mptsc* cannot be modified and represents the maximum security level that can be displayed on a particular terminal.

See sections 2.2, 2.3, 3.1, 4.18, 5.1.

- *cptsc* is initially set to  $L$  for all terminals.
- Processes can access, unless from the MLS model point of view, any object. This means that the system will not prevent processes (no matter on behalf of whom they are acting) from reading objects with any access class. Belive it or not but this feature by itself is not insecure.

See sections 4.12, 4.13, 4.9.

---

<sup>1</sup>We have included other security requirements but in this section we only comment about the most important ones.

- But, processes cannot write information in lower objects once they have read higher objects. In other words, a process can read any file, taking highly classified information into its memory space, but it will not be able to write this information from its memory into lower level files.

This will be implemented by moving the control of access from `open` to `read` and `write`.

You may wonder, what this feature does for the user? The answer is simple: a user may edit two different files with distinct access classes at the same screen and at the same moment. If a BLP-like model is implemented, this situation cannot happen. In our model, the only thing that is forbidden to the user is to save the lower level file being edited. This is so because the process could have read data from the higher level file and be willing to write it to the lower level file.

See sections [4.17](#), [4.18](#), [4.9](#).

- Objects created with `creat`, `mkdir` or `pipe` will have *L* as their initial access class, and it will be increased to the access class of the first chunk of information that is written into them. But, once these objects are not empty their access classes cannot be modified (except by the security administrator).

See sections [4.4](#), [4.2](#).

- If an object is emptied (for example issuing `truncate` over it), then its access class can be modified by a user or the system. The user can change the access class of an empty object by executing `chobjsc`; the system changes the access class of an empty object when new data, with a different access class, is written into the object.

See sections [4.1](#), [4.17](#).

- Process initiated by trusted programs with `execve` have their memory spaces classified at *L*.

See section [4.5](#).

### 1.3.2 Functional and Security Requirements

The model presented in this document is an abstraction of the real Linux kernel. Care must be taken when implementing it as a modification of the Linux kernel, because some features of the kernel has been specified in such a way that differ from its actual implementation. This is so because models do not give all the details, but not because the model is saying “implement this or that in the way it is specified”.

Thus, programmers are faced with a model that in one hand describes properties that must be implemented as they are described; and on the other hand, it describes properties that are already implemented and which implementation should not be changed.

Roughly speaking, properties that must be implemented as specified are those that deal with information flow or access control. And properties that must be left unchanged are all others. However, to determine whether a predicate is specifying a security property or not, it is not always easy. Predicates involving the variables listed below (see chapter [2](#) for details) are not, in most cases, security predicates:

- *input, output, ready*
- *stdin, stdout*
- *ltcont*
- *objs, ocont*
- *mem*

But, if other variables are involved in the same predicate, it is likely that this predicate is specifying a security property. Take as an example the specification of the **read** system call, that is the operation named *Read* on page 49. There, you will find that state variable *mem* is updated by adding to it some part of *ocont o?* -i.e. a process reads some part of *o?*'s content and the system put it in the process' memory. This predicate is not a security predicate. It is a functional predicate: it says what happens when a process reads a file. However, this is a very important property of *Read*: in fact we took many precautions in specifying this operation because of this functionality. On the other hand, predicates:

$$INF\{o : OBJECT \mid o \in (aprocs\ pid?).mmfw \bullet osc\ o\} \succeq osc\ o?$$

and

$$supr' = Sup\ supr\ sc \wedge sc = (osc\ o?)$$

are security predicates. You can distinguish them because they mention *osc* and *supr* which are new state variables -i.e. variables not present in the Linux kernel.

## 1.4 Style conventions

We have made a great effort to keep a uniform structure for identifiers. Our conventions are as follows:

- Basic types are uppercase, like *CATEGORY*, and in the singular
- Only the first letter of each word in schema names is uppercase, like *SecClass*, and if it is a state schema its name is in the singular
- Elements of enumerations are in lowercase, like *undef*
- Variables are in lowercase, like *level*

Each operation schema is divided into a number of schemas. There is one schema for each successful case, and one schema for each unsuccessful case. A schema, called *Okschema*, is defined as the disjunction of all schemas representing successful cases; and another schema, called *Eschema*, is defined as the disjunction of the unsuccessful cases. If there is only one schema for successful or unsuccessful cases, then only the *Okschema* and the *Eschema* are defined. In other words, always there must be an *Ok*-schema and an *Eschema*. Finally, the operation schema is defined as the disjunction of the *Okschema* and the *Eschema* (called *Tschema*). So we have:

$$\begin{aligned} Okschema &\triangleq SuccessfullCase_1 \vee \dots \vee SuccessfullCase_n \\ Eschema &\triangleq UnsuccessfullCase_1 \vee \dots \vee UnsuccessfullCase_m \\ Tschema &\triangleq Okschema \vee Eschema \end{aligned}$$

We have defined name conventions for all those schemas:

- The name of a *Tschema* starts with an abbreviation of the name of the state schema, followed by the name of the operation, for example *SCGetCat*
- The name of an *Okschema* starts with the name of the corresponding *Tschema* followed by *Ok*, for instance *SCGetCatOk*
- Each of the disjuncts of an *Okschema* has the same name of the *Okschema* followed by a natural number starting at 1, for example *SCSetLevelOk2*
- The name of an *Eschema* starts with the name of the corresponding *Tschema* followed by *E*, for instance *SCGetLevelE*

- Each of the disjuncts of an Eschema has the same name of the Eschema followed by a natural number starting at 1, for example *SCAddCatE1*

We have not used the standard Z style for recording state invariants. Instead, for each state schema we record its invariant as follows:

1. A normalized state schema is defined without any predicate; say its name is *Schema*
2. A state schema named *SInv* is defined by including *Schema* and recording its invariant
3. Operation schemas acting over *Schema* do not include *SInv*, thus all preconditions are explicit
4. For each operation, *Op*, that changes *Schema*, the following proof obligation is written:

$$SInv \wedge Op \Rightarrow SInv'$$

Hence, if a proof is given the invariant is guaranteed and programmers have explicit preconditions to code. For example, consider the following specification where variable  $x$  is intended to be non-negative. We start by defining a state schema where variable  $x$  is normalized and unconstrained.

$X$	
$x : \mathbb{Z}$	

Then, we define a schema capturing the invariants for schema  $X$ .

$XInv$	
$X$	
$x \geq 0$	

Now, we define an operation that could potentially violate the invariant so we include the appropriate precondition.

$Decr$	
$\Delta X$	
$x > 0$	
$x' = x - 1$	

Finally, a proof obligation is introduced in order to guarantee that  $XInv$  is indeed an invariant.

**theorem** DecrPI  
 $XInv \wedge Decr \Rightarrow XInv'$

Were we defined  $X$  as follows:

$X$	
$x : \mathbb{Z}$	
$x \geq 0$	

we could have defined  $Decr$  to be

$Decr$	
$\Delta X$	
$x' = x - 1$	

because the invariant is verified by definition. Hence, we left  $x > 0$  implicit and the specifier or the programmer must make it explicit what is equivalent to the first approach.

At the end of this document you may find an index listing all the formal terms defined and the page number where its definition is. We believe this index will be of great help because you may find terms quickly. Sadly, page numbers may be off by one due to syntactical restrictions imposed by Z/EVES.

Any word written in **typewriter type style** refers to program code in the Linux kernel, operating system commands or the like.

## Chapter 2

# The State of the File System

The title of this section is a little bit confusing because as it is suggested in [13], the engineer should never describe the state of the machine when doing requirements engineering. Instead, he or she must describe the state of the environment and the behavior of the machine connected to this environment. In this case the machine is the file system portion of the operating system kernel.

We think of a server running the operating system connected to a number of physical terminals through point-to-point serial lines. Each of them interfaces with the system through a logical terminal which is a software device. There must be a one to one relation between physical and logical terminals that cannot be re-defined (except with a controlled procedure). This means that the binding between logical terminals and physical ports (on the server) cannot be changed by any user. We think that this kind of changes are seldom needed.

Physical terminals are, basically, *dummies*, i.e. they have no processing capabilities. If a physical terminal performs some computation then it is assumed to be correct and trustworthy. More precisely, each of them comprises a keyboard and a screen, and a point-to-point cable that connects each of them with the computer. Physical terminals include printers. If a physical terminal contains a second storage unit, then the system (running on the server) is not responsible for the data stored in that unit, and we assume that nothing can save protected information in that unit -however, we allow the server to read information stored in it.

The server contains a hard disk which in turn contains the information to be protected. It should be noted that both the hard disk and its data are part of the environment -i.e. they are not the operating system program. Also, there are some individuals whom will use the system from the physical terminals. The system needs a way to recognize these persons and their security clearances, then there is a data base containing all this information -which in turn is stored in the server's hard disk and must be protected.

To summarize the environment comprises:

- A set of physical terminals from which users interact with the system, each terminal has an input and an output device;
- A set of logical terminals or device drivers that mediate the communication with the physical terminals
- An external database of users and their security attributes;
- A set of objects (files and directories) and their security attributes (ACL, access classes); and
- A set of processes acting on behalf of users which are the only ones that can invoke file system services.

Each of these sets will be modeled as one or more state schemas as it is shown below. The invariants of these state schemas are defined in chapter 7.

State variables can be of one of three kinds:

- machine controlled,
- environment controlled, or
- machine and environment controlled.

If a variable is controlled by the machine, then the machine is the only entity that can modify its value and the environment cannot modify it in any way specified or not. The same is true for environment controlled variables. Variables controlled by both machine and environment can be modified by either of them. Thus, we will indicate, in so called *control tables*, what variables are machine or environment controlled. Machine controlled variables must be implemented inside the TCB (see section 2.1) given that it is the only one component we can trust.

## 2.1 Trusted Computer Base

We are modeling a secure computer system or, better, a system that should be secure against a particular kind of threat. Hence, we need to state precisely which entities are trusted and which are not. The system, composed of the file system program, the set of physical terminals, and all the hardware needed to store the external databases, are trusted. Also, some special programs (such as `login`, `init`, etc.) are considered trusted processes. Particularly, physical terminals are trusted as much as their security class suggests. Also, every user is trusted as much as his or her security class. Programs and processes are not trusted, every one of them may be a Trojan horse.

Trusted programs must be guarded against unauthorized modifications; we include in this category the operating system program. This is hard to achieve in UNIX-like operating systems. We assume hardware is protected against unauthorized modifications with physical security countermeasures.

See [6] for more details about the TCB.

## 2.2 Physical Terminals

Terminals have one input device and one output device. Users enter data through the input device and see data (sent by the system) on the output device. We will assume that terminals read and write characters. Each terminal has a maximum security class depending on things such as location, who can be seated in front of it, etc. But users can enter and see information at various levels (access classes). Hence, it is necessary that the system be informed about the classification of the input it is receiving in every moment. This access class is called *current access class*. Thus, if 'a' is a character entered at a particular classification, then another 'a' entered at another access class must be regarded as fundamentally different than the first one. In fact, a basic property this system must enforce is that any input done at a particular classification must not be outputted at a lower classification. In other words, if 'a' is entered through a terminal working at current access class  $c_i$ , then it cannot be written (in any future time) on a terminal working at current access class  $c_o$ , with  $c_o \prec c_i$ .

In consequence, characters must be qualified by the current access class of the terminal through which they are entered. This does not mean that an actual implementation of the system has to keep this information. Its only purpose is to enable us to formally prove that the model verifies the property stated above. We model qualified characters as follows:

**Z Section** *state*, **parents:** *sc, subjectsc, acl*

*CHAR*  $\approx$  elements of this set are inputed to or outputed from the system

[*CHAR*]

*CCHAR* == *CHAR*  $\times$  *SecClass*

We think of physical terminals as character devices, i.e. characters are entered or outputed one at a time. Thus, we model physical terminals as composed of two *CCHAR* variables, one represents the input device and the other the output device.

*input*  $\approx$  character already entered but yet not procesed by the system

*output*  $\approx$  last character written by the system on the terminal screen

*ready*  $\approx$  is used to sicronize the physical terminal with the logical terminal attached to it; if *ready* is greater or equal to zero then *input* must be processed, otherwise it has already been processed

*PhysicalTerminal*

*input* : *CCHAR*

*output* : *CCHAR*

*ready* :  $\mathbb{Z}$

Next is the initial schema for a *PhysicalTerminal*. Initially every physical terminal has no input to process and its output device is empty. To model this we use a special character called *null*. This character cannot be printed nor processed.

| *null* : *CHAR*

*PTInit*  $\triangleq$  [*PhysicalTerminal*; *SCInit* | *input* = *output* = (*null*,  $\theta$ *SecClass*)  $\wedge$  *ready* < 0]

We identify each physical terminal connected to the system with an element of *PTERM*.

*PTERM*  $\approx$  identifiers for physical terminals

*mptsc*  $\approx$  the maximun security class of each physical terminal

[*PTERM*]

| *mptsc* : *PTERM*  $\rightarrow$  *SecClass*

We model the maximun access class of a physical terminal as an axiomatic definition (and not as a part of the state) because we do not model operations that change this attribute.

In a particular moment there are some users working on some physical terminals at a particular classification. We include the bijection from *PTERM* onto *PhysicalTerminal* as part of the state because the state of each *PhysicalTerminal* may change over time, and not because the relation between *PTERM* and *PhysicalTerminals* is intended to change.

*pts pti*  $\approx$  is the state of the physical terminal identified with *pti*



$upt\ u \approx$  the physical terminal through which user  $u$  is working

$cptcs\ pt \approx$  current classification of physical terminal  $pt$ ; it is not a partial function because every physical terminal has a current security class despite a user is working on it or not

<i>UsersAndTerminals</i>
$pts : PTERM \rightarrow PhysicalTerminal$
$upt : USER \rightarrow PTERM$
$cptsc : PTERM \rightarrow SecClass$

The system starts in a state where no users are working on it, and the current access class of each terminal equals  $L$ . Also, each physical terminal connected to the system is in its initial state.

<i>UTInit</i>
<i>UsersAndTerminals</i>
<i>SCInit</i>
<i>PTInit</i>
$ran\ pts = \{\theta PhysicalTerminal\}$
$upt = \emptyset$
$ran\ cptsc = \{\theta SecClass\}$

Table 2.1 is the control table for the state variables defined in this section.

Variable	Machine	Environment
<i>input</i>		•
<i>output</i>	•	
<i>ready</i>	•	•
<i>cptsc</i>	•	
<i>upt</i>	•	•
<i>pts</i>		•
<i>mptsc</i>		•

Table 2.1: Control of variables of *PhysicalTerminal* and *UserAndTerminals*

## 2.3 Logical Terminals

Logical terminals are what the system sees of a physical terminal. Logical terminals sometimes are called *ttys*. We are specially concerned with logical terminals, and we assign to them different security properties with respect to other kind of objects. At design or implementation level, other character devices may be considered as logical terminals because we model very abstract operations over them. We think of them as having an identifier, a content, and a security class.

$LTERM \approx$  identifiers of logical terminals of the system

$stdin \approx$  sequence of input that has been taken from the environment, i.e. a physical terminal

$stdout \approx$  sequence of output written by a process and not yet outputted to the medium

[*LTERM*]

<i>LTCont</i>
<i>stdin, stdout</i> : seq <i>CCHAR</i>

Note that a logical terminal buffers the input and output send to and received from the physical terminal attached to it. Initially, the content of a logical terminal is empty.

$$LTCInit \triangleq [LTCont \mid stdin = stdout = \langle \rangle]$$

We need to assign logical terminals to physical terminals. There is exactly one logical terminal for each physical terminal and viceversa. Hence, we define:

*ltcont* *lt*  $\approx$  the content of logical terminal *lt*

*ltsc* *lt*  $\approx$  the security class of logical terminal *lt*

*ttys* *lt*  $\approx$  the physical terminal connected to logical terminal *lt*. We assume that if logical terminal *lt* is connected to physical terminal *pt*, then every input received by *lt* can only come from *pt*, and every output send by *lt* is written only on *pt*.

<i>LogicalTerminals</i>
<i>ltcont</i> : <i>LTERM</i> $\rightarrow$ <i>LTCont</i>
<i>ltsc</i> : <i>LTERM</i> $\rightarrow$ <i>SecClass</i>
<i>ttys</i> : <i>PTERM</i> $\rightarrow$ <i>LTERM</i>

Initially logical terminals are empty, and the access class of each of them equals *L* (which is equal to the current access class of the physical terminal connected to it).

<i>LTCInit</i>
<i>LogicalTerminals</i>
<i>LTCInit</i>
<i>SCInit</i>
ran <i>ltcont</i> = { $\theta LTCInit$ }
ran <i>ltsc</i> = { $\theta SecClass$ }

Table 2.2 is the control table for the state variables defined in this section.

Variable	Machine
<i>stdin</i>	•
<i>stdout</i>	•
<i>ltcont</i>	•
<i>ltsc</i>	•
<i>ttys</i>	•

Table 2.2: Control of variables of *LogicalTerminals*

### 2.3.1 Design and Implementation Comments Regarding Terminals

As shown in Tables 2.1 and 2.2 *mptsc* is controlled by the environment, and *ttys* and *ltsc* are controlled by the system. Moreover, strictly speaking *mptsc* is not shared between the system and the environment. All this information has to be considered at the implementation level.

First, the system has to have a way to know the access class of each physical terminal connected to it because many decisions are taken based on these values. Given that there is a one to one relation between logical and physical terminals, we suggest to include this value in the i-node of each logical terminal. Thus, logical terminals (in fact their i-nodes) will store two access classes: one representing *mptsc* and the other *ltsc*.

There is no need in storing *cptsc* explicitly in the i-nodes of logical terminals. The reason is that, as schema *SFSInvPLT* shows (see chapter 7), the following predicate must be a system invariant:

*The current access class of a physical terminal must be equal to the access class of the logical terminal attached to it.*

Thus, at implementation level, having a field for *ltsc* is enough to represent both access classes because initially they are equal and they are always changed at the same time and to the same value.

Second, as we have said the system has no way to know the access class of a physical terminal except by looking at the access class of its logical terminal and by assuming they are equal. But this is clearly an assumption not completely controlled by the system. For example, if a clerk decides to move a terminal to a public place (thus, implicitly changing its access class), the system will not perceive that change and will write classified information on it when requested. Hence, care must be taken when access classes of both physical and logical terminals are changed because otherwise it could be possible for attackers to see classified information.

Third, we tried to describe the transfer of characters between a physical terminal and the system. This transfer is made on a character by character basis. Every time a user pushes a key, an interruption occurs and the kernel copies the corresponding character into the logical terminal connected to the keyboard. In this way, the next time a process reads from its tty this character may be available. Clearly, the interruption acts as a sincronizing event between two processes: the system and the environment.

Z does not directly support sincronization between concurrent processes and we think that it is not necessary to model this feature. Hence, we use a shared variable, *ready*, to sincronize system and environment. However, with this mechanism some input may be lost. On the other hand, we try to model a security problem and not a concurrency problem, so we do not care about lost characters (they do not affect confidentiality). In summary, we have modeled the communication between physical and logical terminals just in order to be able to reason about security; programmers must leave the actual implementation as it is whenever it does not conflict with the intended security properties.

Note that the initial current access class of physical terminals is *L*. This means that access classes of logical terminals are also *L*. Thus, initially, every input sent to the system will be classified at *L*. We believe this is a good policy that helps to increase the usability of the system. MLS systems, quite naturally, tend to upgrade information, hence we decided to start the system and the user activity as lower as possible. Users must be warned of this fact.

Fourth, there is exactly one logical terminal for each physical terminal and viceversa (this is codified in the type of *ttys*). To preserve this relation is paramount to system security. When we said that there is only one logical terminal for each physical terminal we mean that whatever is written in or read from a particular logical terminal, goes to or comes from its physical terminal, and nowhere else. This must be true even for processes run by *root*. Moreover, all this activity must be controlled by the system. That is, processes can only read or write from logical terminals and they never can get direct access to physical terminals.

Particularly, it must be impossible for a process (even for a `root` process) to connect a logical terminal to a physical terminal. Process are part of the environment, and, as Table 2.2 shows, state variable *ttys* is controlled by the machine (i.e. the operating system kernel program). Hence, were a process been able to connect a logical terminal to a physical terminal, then the environment would have been in control of *ttys*. More precisely, the machine is in control of *ttys* if the relation between logical and physical terminals is codified in the operating system kernel program, and there is no way (with or without system calls) for any process to alter this relation. Clearly, this implies that the only way to modify this relation is by recompiling the kernel program.

## 2.4 Users Allowed to Work on the System

As we said in the introduction of this chapter, part of the environment is a database of users and their security attributes. *UserSecClass* is defined in chapter 8 (it models the relationship between users and security classes).

*users*  $\approx$  set of users allowed to use the system

*grps*  $g \approx$  the set of users that belong to group  $g$

<i>Users</i>
<i>UserSecClass</i>
<i>users</i> : $\mathbb{P} \text{ USER}$
<i>grps</i> : $\text{GRPNAME} \leftrightarrow \mathbb{P} \text{ USER}$

In the initial state *Users* contains a few built-in users and groups, and standard access classes for them.

*root*  $\approx$  the standard UNIX administrator; in GTL it will only be able to manage ACLs but not access classes

*allgrp*  $\approx$  a group that contains all users

*rootgrp*  $\approx$  a group of administrators that is equivalent to *root*

*secadm*  $\approx$  is designated in chapter 8

<i>root</i> : <i>USER</i>
<i>allgrp, rootgrp</i> : <i>GRPNAME</i>

  

<i>UInit1</i>
<i>Users</i>
<i>users</i> = { <i>root, secadm</i> }
<i>grps allgrp</i> = { <i>root, secadm</i> }
<i>grps rootgrp</i> = { <i>root</i> }
<i>SECADMIN</i> $\notin$ ( <i>usc root</i> ). <i>categs</i>

$$UInit \cong UInit1 \wedge USCInit$$

Table 2.3 is the control table for the state variables defined in this section.

Variable	Machine
<i>users</i>	•
<i>grps</i>	•
<i>usc</i>	•

Table 2.3: Control of variables of *Users*

## 2.5 Protected Objects

The objects of that the system must protect include files, directories, pipes, and so on (see [6] for more details).

*OBJECT*  $\approx$  the set of protected objects, i.e. regular files, directories, pipes, etc. An element of this set represents the absolute path or name of the object.

*OCONT*  $\approx$  objects store a sequence of classified characters

$[OBJECT]$

$OCONT == \text{seq } CCHAR$

Next, we model the object database and all their security attributes.

*objs*  $\approx$  the set of objects that currently exist in the system

*ocont*  $o \approx$  the content of object  $o$

*oacl*  $o \approx$  the access ctrl list of object  $o$

*osc*  $o \approx$  the security class of object  $o$

*FileSystemObjects*

$objs : \mathbb{P} OBJECT$

$ocont : OBJECT \leftrightarrow OCONT$

$oacl : OBJECT \leftrightarrow AccessCtrlList$

$osc : OBJECT \leftrightarrow SecClass$

As we said in section 2.1, the TCB is composed of hardware and software. The software portion of the TCB is composed of programs (trusted processes) and data files. We model the software portion of the TCB with the following global variable.

$| \text{soft tcb} : \mathbb{P} OBJECT$

We consider *soft tcb* as a global entity and not part of the state because we are not interested in operations that modify this set.

Initially, the object database contains only the software portion of the TCB.

$FSOInit \triangleq [FileSystemObjects \mid objs = \text{dom } osc = \text{dom } oacl = \text{dom } ocont = \text{soft tcb}]$

Table 2.4 is the control table for the state variables defined in this section.

Variable	Machine
<i>obj</i>	•
<i>ocont</i>	•
<i>oacl</i>	•
<i>osc</i>	•
<i>softtcb</i>	•

Table 2.4: Control of variables of *FileSystemObjects*

### 2.5.1 The Access Class of Directories

Directories contain file names. File names may be important in themselves or not. For example, if you have file *mycompetitor* stored in a directory, that name conveys some information to an attacker. Instead, if you name the same file *xr56wT* no one (program or person) may deduce anything about its content. Now, what should be the access class of the directory where *mycompetitor* is stored? What should it be if you name the file *xr56wT*?

We think that a directory must have an access class different from  $L$  if and only if any of its file names can give some information to an attacker. Moreover, keeping the access class of directories close to  $L$  will increase the usability of the system without necessarily reducing its security. In particular, we suggest that users' home directories be classified at  $L$ .

## 2.6 Processes

We consider that a process records the following information:

*usr*  $\approx$  the user who launched the process; it is equal to the `uid` field of the `task_struct`

*suid*  $\approx$  it is possible that a process temporarily changes its identity (cf. SUID), this temporal identity is stored in *suid*; most of the time *usr* = *suid*; this alternative identity is used to check DAC rights; it is equal to the `euid` or `fsuid` fields of the `task_struct`<sup>1</sup>

*lt*  $\approx$  the identifier of the logical terminal associated with the process; i.e. the process reads from and writes to *lt* to interact with the user<sup>2</sup>

*or*  $\approx$  the objects that the process has opened for reading and not yet closed

*ow*  $\approx$  the objects that the process has opened for writing and not yet closed

*mmfr*  $\approx$  memory mapped files in *read* mode (see `mmap` manual page)

*mmfw*  $\approx$  memory mapped files in *write* mode (see `mmap` manual page)

*supr*  $\approx$  the least upper bound of the access classes of the information that the process has read (it does not matter if the files from which this information comes from are closed); this variable is called *working access class* of the process, we will talk of a process working at a particular access class when its working access class is that access class

*mem*  $\approx$  the state of its memory space; it is intended to represent that portion of the process' memory that holds data taken from the environment (i.e. objects and devices)

<sup>1</sup>Usually `euid` and `fsuid` are equal, even when the process is running SUID to other user.

<sup>2</sup>This is an unnecessary restriction of the model that will be removed in future versions; that is, a process can interact with many logical terminals.

$prog \approx$  the program from which the process was built by the system

<i>Process</i>
$usr, suid : USER$ $lt : LTERM$ $or, ow : \mathbb{P} OBJECT$ $mmfr, mmfw : \mathbb{P} OBJECT$ $supr : SecClass$ $mem : seq CCHAR$ $prog : OBJECT$

A process starts with no open files, its memory is empty, and thus the least upper bound of the information it has read is  $L$ .

<i>PInit</i>
<div> <div><i>Process</i></div> <div><i>SCInit</i></div> </div> $usr = suid$ $or = ow = mmfr = mmfw = \emptyset$ $supr = \theta SecClass$ $mem = \langle \rangle$

At any moment there will be a number of active processes. Each of them it is identified by a process identifier, so we model the set of active processes as a partial function from  $PROCID$  to  $Process$ . Initially this set is empty.

$PROCID \approx$  process identifiers

$PROCID == \mathbb{N}$

<i>ProcessList</i>
$aprocs : PROCID \rightarrow Process$

$PLInit \triangleq [ProcessList \mid aprocs = \emptyset]$

Table 2.5 is the control table for the state variables defined in this section. Note that, except for  $mem$ , every state variable of a process is controlled by the system. This means that they must be implemented inside the kernel because otherwise a process could modify them without telling to the operating system. In a benign environment these variables could be implemented anywhere.

## 2.7 The State of the Environment

We summarize the state of the environment and its initial state in a couple of schemas.

<i>SecureFileSystem</i>
$UsersAndTerminals$ $LogicalTerminals$ $Users$ $FileSystemObjects$ $ProcessList$

Variable	Machine	Environment
<i>usr</i>	•	
<i>suid</i>	•	
<i>lt</i>	•	
<i>or</i>	•	
<i>ow</i>	•	
<i>mmfr</i>	•	
<i>mmfw</i>	•	
<i>supr</i>	•	
<i>mem</i>	•	•
<i>apro</i>	•	

Table 2.5: Control of variables of *Process* and *ProcessList*

<i>SFSInit</i>	
<i>UTInit</i>	
<i>LTInit</i>	
<i>UInit</i>	
<i>FSOInit</i>	
<i>PLInit</i>	

end of Z Section *state*



## Chapter 3

# Operations Controlled by the User

We divided the operations into three disjunct groups:

- Operations controlled by users
- Operations controlled processes
- Operations controlled by the system

This chapter includes the formal specification of operations that belongs to the first group, the next two chapters include operations defined in the remaining groups. In any of these three chapters, operations are ordered alphabetically, one for section. Next, we make some general comments regarding operations included in any group.

We have formalized those operations we believe are security relevant. Also we have specified operations that complete the model (for example, *Close* is not security relevant but completes the model in some sense). Moreover, for each operation we tried to formalize just those features related to security and not features purely functional.

Operation can be *environment controlled* or *system controlled* [13]. Environment controlled means that the environment initiates the execution of the operation; it does not mean that the environment necessarily performs the execution (for example, *Open* is initiated by the environment but performed by the system, in other words *Open* is not spontaneously executed by the system). System controlled means that the system initiates the operation. An environment controlled operation can be controlled by only one of user or process. System controlled operations are not system calls.

In the description of each operation we have included comments to help programmers to do their jobs. We tried to structure comments to ease reading but we were not rigid. Every section starts with a list with the following items:

**Description** A one line description of the operation functionality

**Input parameters** A list of named input parameters along their types (all input parameters names end with a question mark, “?”)

**Kinds of objects** This is an optional item. It appears only when at least one input parameter is of type *OBJECT*. In this case this item indicates, for each input parameter of that type, to what kinds of objects the operation could be applied. For example, *Write* writes into files and not into directories, so this item will say “Files”.

**Preconditions** An informal description of the preconditions of the operation

**Postconditions** An informal description of the postconditions of the operation

After this list there is a more detailed comment about the purpose and functionality of the operation.

Mixed with the formal text there are as many comments as we judged necessary to make the specification understandable. Preceding every schema representing a successful case there is a comment explaining it. Error schemas are seldom explained.

Some operations end with a subsection containing a brief or detailed account of design or implementation considerations.

We strongly encourage to read all the section before start implementing the respective operation. Moreover, a somewhat conscious reading of the entire document worth the time spent on it.

### 3.1 Chinsc

**Description** Changes the security class of the input entered by the user

**Input parameters**  $pt? : PTERM$ ;  $sc? : SecClass$

**Preconditions**  $pt?$  must be being used, that is a user must be logged on  $pt?$ ; the maximum access class of  $pt?$  must dominate  $sc?$ ; and the input buffer of the logical terminal connected to  $pt?$  must be empty

**Postconditions**  $sc?$  is the new current access class of  $pt?$ , and the new access class of the logical terminal connected to  $pt?$

This operation will be used by users to communicate to the system that they want to change the classification of their input. Once a user execute *Chinsc* the system will classify the input entered by the user from this time on at the access class indicated by him.

The intention behind this operation is to increase the level of usability of the system. We believe that it will allow users writing confidential information to start writing public information with a few keystrokes, and viceversa. Moreover, they will be able to do this on the same terminal and without leaving and reentering into the system.

**Z Section** *chinsc*, **parents:** *state, definitions*

This operation has one successful case. The first precondition says that  $pt?$  must be being used; that is, a user must be logged on  $pt?$ .

The second precondition is tantamount to system security because it imposes a bound to the classification of the information that can be entered on this particular terminal. Hence, the system prevents inattentive users to disclose information by typing it in a terminal not trusted enough. Think of a terminal close to a window and an attacker watching the keyboard.

The last precondition ensures that there are no input to process in the logical terminal connected to  $pt?$ . This allows more simple implementations given the fact that the implemented system will not handle *CCHARs* but *CHARs* which carry no security labels. In this way, it is impossible to merge inputs with different classifications.

$ \begin{array}{l} \textit{ChinscOk} \\ \Delta \textit{UsersAndTerminals} \\ \Delta \textit{LogicalTerminals} \\ \Xi \textit{Users}; \Xi \textit{FileSystemObjects}; \Xi \textit{ProcessList} \\ pt? : PTERM \\ sc? : SecClass \\ repo! : SFSREPORT \end{array} $
$ \begin{array}{l} pt? \in \text{ran } \textit{upt} \\ (mptsc \ pt?) \succeq sc? \\ (ltcont \ (ttys \ pt?)).stdin = \langle \rangle \\ cptsc' = cptsc \oplus \{pt? \mapsto sc?\} \\ ltsc' = ltsc \oplus \{(ttys \ pt?) \mapsto sc?\} \\ pts' = pts \\ ltcont' = ltcont \\ upt' = upt \\ ttys' = ttys \\ repo! = ok \end{array} $

If preconditions are met,  $sc?$  is the new current access class of  $pt?$ , and the new access class of the logical terminal connected to  $pt?$ . Note that changing the access class of the logical terminal is consistent with our policy regarding the modification of security attributes because the input buffer is empty (see section 4.4). Also, pay attention to the fact that the input will be classified at the new access class for every existing and future user's processes reading from  $tty \ pt?$ . In other words, the input will be classified at  $sc?$  until the user leaves the system or invokes the operation once more.

We have considered that requesting a new access class beyond  $mptsc \ pt?$  or requesting a valid new access class but when the input buffer is not empty must have a *permissionDenied* response.

$ \begin{array}{l} \textit{ChinscE1} \\ \Xi \textit{SecureFileSystem} \\ pt? : PTERM \\ sc? : SecClass \\ repo! : SFSREPORT \end{array} $
$ \begin{array}{l} (\neg (mptsc \ pt?) \succeq sc?) \\ \vee (ltcont \ (ttyspt?)).stdin \neq \langle \rangle \\ repo! = \textit{permissionDenied} \end{array} $

The user interface to the trusted path will be some key combination that can be invoked even if there is no user working on that terminal, but in this case the system's response is an error condition. At implementation level, there could be no response.

Finally, *Chinsc* is defined as always.

$$\textit{ChinscE2} \triangleq \textit{TerminalNotUsed}$$

$$\textit{ChinscE} \triangleq \textit{ChinscE1} \vee \textit{ChinscE2}$$

$$\textit{Chinsc} \triangleq \textit{ChinscOk} \vee \textit{ChinscE}$$

end of Z Section *chinsc*

### 3.1.1 Design and Implementation Comments

This operation must be implemented with a trusted path. We decided to use a portion of the screen as a trusted channel reserved for communications originated by the kernel or a trusted process. See [6] for more details.

## 3.2 Input

**Description** A user types in a character in his physical terminal

**Input parameters**  $pt? : PTERM$ ;  $c? : CHAR$

**Preconditions**  $pt?$  is being used by some user

**Postconditions**  $c?$  is classified at the current access class of the physical terminal and a signal is issued warning the system that new input is available on  $pt?$

This operation represents the user typing in a character on his terminal's keyboard. It is completely controlled by the user, i.e. he may type in characters at his will, the system has no way to constrain this action. But, as we have explained in section 5.1 with this model some characters may be lost.

Clearly, this operation must not be implemented; it was included for completeness.

**Z Section** *input*, **parents:** *state, definitions*

We start by defining a schema at the *PhysicalTerminal* level which models how a character is received and classified at some security class ( $sc$ ). This value is further constrained in schema *InputOk*.

<i>PTInput</i>
$\Delta PhysicalTerminal$
$c? : CHAR$
$sc : SecClass$
$input' = (c?, sc)$
$output' = output$
$ready' = 1$

The next schema describes the operation. Note that it does not contain preconditions constraining a user to type in characters; it would be unrealistic to do so. Predicate  $sc = cpts\ pt?$  establishes that the access class at which  $c?$  is classified, is precisely the current access class of the physical terminal at which  $c?$  is received.

$InputOk$ $\Delta UsersAndTerminals$ $\Xi ProcessList; \Xi LogicalTerminals; \Xi Users; \Xi FileSystemObjects$ $PTInput$ $c? : CHAR$ $pt? : PTERM$ $repo! : SFSREPORT$
$pt? \in \text{ran } upt$ $sc = cptsc \ pt?$ $pts' = pts \oplus \{pt? \mapsto \theta PhysicalTerminal\}$ $upt' = upt$ $cptsc' = cptsc$ $repo! = ok$

It is an error to receive an input in a terminal not being used.

$InputE \triangleq TerminalNotUsed$

$Input \triangleq InputOk \vee InputE$

**end of Z Section** *input*

### 3.3 Login

**Description** A user logs in on the system through a particular physical terminal

**Input parameters**  $u? : USER; pt? : PTERM$

**Preconditions**  $u?$  must be a user recognized by the system,  $pt?$  is not being used by any user, and the access class of  $u?$  must dominate the maximum access class of  $pt?$

**Postconditions** A new process is initiated acting on behalf of  $u?$ ; this process is in its initial state (i.e. its memory is empty, has no open files, and executes at the lowest bound of the access class set,  $L$ )

This operation represents the standard **login** program of UNIX-like operating systems, but we have not modeled the authentication mechanism of Linux. We are only interested in access control or information flow, not in authentication mechanisms. However, at implementation level an authentication mechanism does have to be implemented.

We strongly recommend to read *ReadLT*'s description before implementing this operation.

**Z Section** *login, parents: state, definitions*

We start with a schema crafting a new process with adequate values given the user and the terminal where the user is logging in.

$PLogin$ $\Delta Process$ $SCInit$ $u? : USER$ $newlt : LTERM$
$usr' = u?$ $suid' = u?$ $lt' = newlt$ $ow' = \emptyset$ $or' = \emptyset$ $mmfr' = \emptyset$ $mmfw' = \emptyset$ $mem' = \langle \rangle$ $supr' = \theta SecClass$

To be able to log in on the system  $u?$ 's access class must dominate that of  $pt?$ ; otherwise the user is not authorized to log in on that terminal. Also,  $pt?$  must be free, i.e. no user should be working on it. As we have explained above the authentication process is not described: the reader may think that it is encoded in  $u? \in users$ .

$LoginOk$ $\Delta UsersAndTerminals$ $\Delta ProcessList$ $\Xi LogicalTerminals; \Xi Users; \Xi FileSystemObjects$ $PLogin$ $u? : USER$ $pt? : PTERM$ $rep_0! : SFSREPORT$
$u? \in users$ $pt? \notin \text{ran } upt$ $usc\ u? \succeq mptsc\ pt?$ $newlt = ttys\ pt?$ $upt' = upt \oplus \{u? \mapsto pt?\}$ $aprocs' = aprocs \oplus \{\min\{p : PROCID \mid p \notin \text{dom } aprocs \bullet p\} \mapsto \theta Process'\}$ $rep_0! = ok$

Postconditions are simple: a new process acting on behalf of  $u?$  is initiated with process identifier equal to the minimum number not being used<sup>1</sup>; and user  $u?$  is associated with terminal  $pt?$ . Hence, from this time on the new process may request services to the operating system and  $pt?$  cannot be used by other users.

Users not recognized by the system cannot log in on the system, and even if they are valid users they cannot log in on a terminal already in use.

<sup>1</sup>This can be implemented as it is in Linux.

$LoginE1$ $\exists SecureFileSystem$ $pt? : PTERM$ $repo! : SFSREPORT$
$pt? \in \text{ran } upt$ $repo! = terminalAlreadyInUse$

If the access class of  $u?$  does not dominate the access class of  $pt?$  the user is not authorized to log in on the system.

$LoginE2$ $\exists SecureFileSystem$ $u? : USER$ $pt? : PTERM$ $repo! : SFSREPORT$
$\neg usc\ u? \succeq mptsc\ pt?$ $repo! = permissionDenied$

$LoginE3 \triangleq UserNotExist$

$LoginE \triangleq LoginE1 \vee LoginE2 \vee LoginE3$

$Login \triangleq LoginOk \vee LoginE$

**end of Z Section** *login*

### 3.3.1 Design and Implementation Comments

#### Trusted Process

The **login** program is implemented as a user space process. Given that we decided to keep this design, the **login** program must be considered a trusted process. Here, trusted means that it has to behave as intended, and that it has permission to by-pass some MLS controls [9]. A program behaves as specified if it is programmed correctly and if every modification comes from a trusted source.

We believe that our development techniques and process will lead us to a correct version of **login**. More precisely, the programming team in charge of the implementation of *Login* must certify its correctness. Given that we will modify an existent program its code must be thorough reviewed.

To preserve the integrity of the program is not easy in UNIX-like operating systems if Trojan horses are a potential threat. For example, in the standard installation, **login** is owned by **root**; then if a Trojan horse is executed by **root** it can modify **login**'s code making it behave accordingly to the attackers' intentions. In consequence some mechanism or configuration must be carried on in order to protect **login**'s integrity. See [6] for more details.

#### Modifications to the login Program

An important modification to the **login** program (despite those formally specified) is that users' passwords will be stored in a distributed database instead of **/etc/shadow**. In this new database each user password is stored in a file owned only by the user and classified at the user's access class. The reason is that higher users have highly classified passwords which in turn must be stored in highly

classified files. The schema based on `/etc/shadow/` is inconsistent with the MLS philosophy. On the other hand, our scheme does not need a SUID program to change passwords.

The `login` program executes the shell for the user. It is very important, due to usability reasons, that the working access class of the shell process be as lower as possible, ideally  $L$ . The *Exec* operation (section 4.5) specifies that the working access class of processes initiated with `exec` will be  $L$  only if the calling process is trusted.

See [6] for more details.

### Further Notes on Trojan Horses and Passwords

One may argue that if passwords are stored in personal files they can be disclosed by Trojan horses acting on behalf of the respective users. While this is partially true, it is also true that the same could happen if passwords are stored in `/etc/shadow` with Trojan horses run by `root`.

We say that this assertion is partially true because such a Trojan horse can disclose a user password only to other users at the same access class. While this is certainly a security violation, it is not a confidentiality problem.

First let's see why a Trojan horse cannot disclose a password to any user. The reason is simple and tightly coupled with the information flow enforced by the system. If user  $u$  executes Trojan horse  $th$  and it reads  $u$ 's password file, its memory space is classified at the access class of that file<sup>2</sup> and thus  $th$  cannot downgrade the password (see sections 4.12 and 4.17). However,  $th$  can copy the encrypted password into a file at the same access class but owned by other user; then this user can log in as  $u$ . But, even in this case there is no information compromise because this second user had access to the "same" information than  $u$ , before knowing his password<sup>3</sup>.

It should be clear now that the `/etc/shadow/` scheme has the same disadvantages as the one proposed by us, but worsen by the fact that all passwords can be "compromised" or modified by just one Trojan horse executed by only one user (`root`). Moreover, the traditional scheme needs a SUID program to change passwords which is always riskier than a non-SUID one.

On the other hand, without an integrity model is impossible to guarantee that passwords are managed with the trusted commands [4].

## 3.4 The Interface for the User

This section contains a schema defining the interface the user can use.

**Z Section** *ucop*, **parents:** *chinsc*, *input*, *login*

$$UserControlledOperations \triangleq Chinsc \vee Input \vee Login$$

**end of Z Section** *ucop*

---

<sup>2</sup>Which is equal to the access class of  $u$ .

<sup>3</sup>Here "same" means: the same using a Trojan horse because  $u$  may have personal information.



## Chapter 4

# Operations Controlled by Processes

In this chapter we have included those operations that are controlled by processes. We strongly recommend to read the introduction to chapter 3. Process controlled operations are implemented as system calls or process internal actions.

### 4.1 Chobjsc

**Description** Changes the access class of a given object

**Input parameters**  $pid? : PROCID$ ;  $o? : OBJECT$ ;  $sc? : SecClass$

**Preconditions** The process issuing the call must be a trusted process;  $o?$  must be empty or the process issuing the call must be acting on behalf of a MAC administrator

**Postconditions**  $o?$ 's access class is set to  $sc?$

This operation allows a trusted process to change the security class of a given object. Following our policy that empty objects cannot compromise information, our model allows ordinary users to change security classes of empty objects.

**Z Section** *chobjsc*, **parents:** *state, definitions*

The first successful case is when the target object is empty. In this case, any user running a trusted program can change access classes.

*ChobjscOk1*

$\Delta FileSystemObjects$

$\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi ProcessList$

$pid? : PROCID$

$o? : OBJECT$

$sc? : SecClass$

$repo! : SFSREPORT$

$pid? \in \text{dom } aprocs$

$o? \in objs$

$ocont\ o? = \langle \rangle$

$(aprocs\ pid?).prog \in softtcb$

$osc' = osc \oplus \{o? \mapsto sc?\}$

$ocont' = ocont$

$objs' = objs$

$oacl' = oacl$

$repo! = ok$

We consider that a directory is empty when its state is equal to the state immediately after it was created. Thus, this operation can be successfully invoked when a directory contains just '.' and '..'.

The second case documents the possibility offered by the system to MAC administrators to change access classes. As always MAC administrators are users which set of compartments contains the special *SECADMIN* category. Note that, despite the user issuing the call is a trusted user, he or she must be using a trusted program too.

*ChobjscOk2*

$\Delta FileSystemObjects$

$\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi ProcessList$

$pid? : PROCID$

$o? : OBJECT$

$sc? : SecClass$

$repo! : SFSREPORT$

$pid? \in \text{dom } aprocs$

$o? \in objs$

$SECADMIN \in (usc\ (aprocs\ pid?).usr).categs$

$(aprocs\ pid?).prog \in softtcb$

$osc' = osc \oplus \{o? \mapsto sc?\}$

$ocont' = ocont$

$objs' = objs$

$oacl' = oacl$

$repo! = ok$

Ordinary users cannot change the access class of a non empty object, and nobody can change the access class of an object if it is not working from a trusted process.

*ChobjscE1*

$\exists \text{SecureFileSystem}$

$pid? : PROCID$

$o? : OBJECT$

$rep_0! : SFSREPORT$

$pid? \in \text{dom } aprocs$

$o? \in objs$

$(ocont\ o? \neq \langle \rangle \wedge SECADMIN \notin (usc\ (aprocs\ pid?).usr).categs$

$\vee (aprocs\ pid?).prog \notin softtcb)$

$rep_0! = permissionDenied$

$ChobjscE2 \hat{=} PidNotExist$

$ChobjscE3 \hat{=} ObjectNotExist$

$ChobjscE \hat{=} ChobjscE1 \vee ChobjscE2 \vee ChobjscE3$

$ChobjscOk \hat{=} ChobjscOk1 \vee ChobjscOk2$

$Chobjsc \hat{=} ChobjscOk \vee ChobjscE$

**end of Z Section** *chobjsc*

## 4.2 Chsubsc

**Description** Changes the access class of a given user

**Input parameters**  $pid? : PROCID$ ;  $u? : USER$ ;  $sc? : SecClass$

**Preconditions** The process issuing the call must be a trusted process;  $u?$  cannot be logged on the system, and the process issuing the call must be acting on behalf of a MAC administrator

**Postconditions**  $u?$ 's access class is set to  $sc?$

This operation allows a trusted process to change the security class of a given user.

**Z Section** *chsubsc*, **parents:** *state, definitions*

Only MAC administrators working from a trusted process can change the access class of a user currently not logged on the system ( $u? \notin \text{dom } upt$ ). We ask that  $u?$  is not logged on the system because, if his or her access class is downgraded then, it might be possible for he or she to temporarily see information for which he or she no longer has the proper authorization.

$ \begin{array}{l} \text{ChsubscOk} \\ \hline \Delta Users \\ \Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi FileSystemObjects; \Xi ProcessList \\ pid? : PROCID \\ u? : USER \\ sc? : SecClass \\ repo! : SFSREPORT \\ \hline u? \in users \\ SECADMIN \in (usc (aprocs pid?).usr).categs \\ (aprocs pid?).prog \in softtcb \\ u? \notin \text{dom } upt \\ usc' = usc \oplus \{u? \mapsto sc?\} \\ users' = users \\ grps' = grps \\ repo! = ok \end{array} $
---

If an ordinary user or a MAC administrator working from a non trusted process request this operation then, the system must return an error condition. Likewise, if the target user is working on the system then the operation is forbidden.

$ \begin{array}{l} \text{ChsubscE1} \\ \hline \Xi SecureFileSystem \\ pid? : PROCID \\ u? : USER \\ repo! : SFSREPORT \\ \hline (SECADMIN \notin (usc (aprocs pid?).usr).categs \\ \vee (aprocs pid?).prog \notin softtcb \\ \vee u? \in \text{dom } upt) \\ repo! = permissionDenied \end{array} $
---

$\text{ChsubscE2} \triangleq \text{PidNotExist}$

$\text{ChsubscE3} \triangleq \text{UserNotExist}$

$\text{ChsubscE} \triangleq \text{ChsubscE1} \vee \text{ChsubscE2} \vee \text{ChsubscE3}$

$\text{Chsubsc} \triangleq \text{ChsubscOk} \vee \text{ChsubscE}$

end of Z Section *chsubsc*

### 4.3 Close

**Description** Closes an object

**Input parameters**  $pid? : PROCID; o? : OBJECT$

**Kinds of objects** It depends on what particular system call *close* represents. If it is `close`, then  $o?$  is a file; if it is `closedir`, then  $o?$  is a directory

**Preconditions** The standard Linux checks

**Postconditions**  $o?$  is removed from the list of open files of  $pid?$

This operation represents system calls such as `close` or `closedir`. It is not necessary to change its actual implementation; it was included in the present document for a matter of completeness.

## Z Section *close*, parents: *state, definitions*

Schema *PClose* sets the state of a process after closing a file. Note that its working access class (*supr*) is not changed (because the process may have copied the entire file in its memory space).

<i>PClose</i>
$\Delta Process$
$o? : OBJECT$
$or' = or \setminus \{o?\}$
$ow' = ow \setminus \{o?\}$
$mmfr' = mmfr$
$mmfw' = mmfw$
$usr' = usr$
$suid' = suid$
$supr' = supr$
$mem' = mem$
$lt' = lt$
$prog' = prog$

An object can be closed only if the requesting process has opened it.

<i>CloseOk</i>
$\Delta ProcessList$
$\exists UsersAndTerminals; \exists LogicalTerminals; \exists Users; \exists FileSystemObjects$
<i>PClose</i>
$pid? : PROCID$
$o? : OBJECT$
$rep_0! : SFSREPORT$
$pid? \in \text{dom } aprocs$
$o? \in (aprocs \text{ } pid?).or \cup (aprocs \text{ } pid?).ow$
$(aprocs \text{ } pid?) = \theta Process$
$aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}$
$rep_0! = ok$

Errors are very simple to deserve a deeper explanation.

<i>CloseE1</i>
$\exists SecureFileSystem$
$pid? : PROCID$
$o? : OBJECT$
$rep_0! : SFSREPORT$
$pid? \in \text{dom } aprocs$
$o? \notin (aprocs \text{ } pid?).or \cup (aprocs \text{ } pid?).ow$
$rep_0! = objectIsNotOpen$

$CloseE2 \hat{=} PidNotExist$

$CloseE \hat{=} CloseE1 \vee CloseE2$

$Close \hat{=} CloseOk \vee CloseE$

**end of Z Section** *close*

## 4.4 Create

**Description** Creates a new object assigning some DAC permissions to it

**Input parameters**  $pid? : PROCID$ ;  $o? : OBJECT$ ;  $perm? : OWN \times GRPP \times OTHP$

**Kinds of objects** It depends on what particular system call *Create* represents. If it is **creat**, then  $o?$  is a file; if it is **mkdir**, then  $o?$  is a directory; if it is **pipe**, then  $o?$  is a pipe

**Preconditions** The same of the standard Linux system call plus the access class of  $o?$ 's parent directory must dominates the access class of the process calling the function

**Postconditions** The same of the standard Linux system call plus  $o?$ 's access class is set to  $L$

This operation represents the standard Linux system call **creat** but also it must be used to implement **mkdir** as is described below.

**Z Section** *create, parents: state, definitions*

We need to define 2-dimensional vectors of *PERM* in order to represent the standard Linux mode for files. We do not define 3-dimensional vectors because we do not consider *execute* as a meaningful permission, actually we equate it with *read*. *OWNP*, *GRPP*, and *OTHP* are used to store owner's, group's and other's permissions, respectively. *ACLSetOGO* fills an *AccessCtrlListData* instance with permissions for one user and two groups. Latter, this schema is used to set permissions for the user who execute the operation, the file's group and the group formed with all the users of the system.

$OWNP == PERM \times PERM$

$GRPP == PERM \times PERM$

$OTHP == PERM \times PERM$

$ACLSetOGO \hat{=}$

$ACLSetModeUsr[u/u?, up/P?]$

$\S ACLSetModeGrp[g/g?, gp/P?]$

$\S ACLSetModeGrp[o/g?, op/P?]$

$\S ACLSetModeGrp[r/g?, rp/P?]$

The first successful case in creating an object is when the object does not exist. Note that we request that the access class of  $o?$ 's parent directory be higher than the access class of  $pid?$ . This is necessary to avoid Trojan horses using low level directories as high bandwidth cover storage channels. In other words, if we do not check this situation it could be possible to create files or directories with names taken from the content of higher files. This restriction may have a severe impact on system usability.

*parentDir*  $o?$  may not belong to  $\text{dom } osc$  but we do not check this explicitly because this is a standard constrain in Linux. This does not mean that this checks should be removed from the implementation. Predicate:

$$\begin{aligned}
u &= (aprocs \text{ pid?}).suid \wedge g = primaryGrp \ u \wedge o = allgrp \wedge r = rootgrp \\
\wedge up &= \{perm?.1.1, perm?.1.2\} \wedge gp = \{perm?.2.1\} \wedge \\
op &= \{perm?.3.1, perm?.3.2\} \wedge rp = \{OWNER\}
\end{aligned}$$

is used to connect variables in *ACLSetOGO* to variables in schema *CreateOk1*.

This case must be used to guide the implementation of `mkdir`; the next case must not be used because `mkdir` does not behave that way.

<div> <div>CreateOk1</div> <div> <math>\Delta FileSystemObjects</math>  <math>\Delta ProcessList</math>  <math>\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users</math>  <math>SCInit</math>  <math>ACLSetOGO</math>  <math>POpenWrite</math>  <math>pid? : PROCID</math>  <math>o? : OBJECT</math>  <math>perm? : OWNP \times GRPP \times OTHP</math>  <math>repo! : SFSREPORT</math> </div> </div>	<div> <math>pid? \in \text{dom } aprocs</math>  <math>o? \notin objs</math>  <math>osc \ (parentDir \ o?) \succeq (aprocs \ pid?).supr</math>  <math>(u = (aprocs \ pid?).suid</math>  <math>\quad \wedge g = primaryGrp \ u</math>  <math>\quad \wedge o = allgrp</math>  <math>\quad \wedge r = rootgrp</math>  <math>\quad \wedge up = \{perm?.1.1, perm?.1.2\}</math>  <math>\quad \wedge gp = \{perm?.2.1\}</math>  <math>\quad \wedge op = \{perm?.3.1, perm?.3.2\}</math>  <math>\quad \wedge rp = \{OWNER\})</math>  <math>(aprocs \ pid?) = \theta Process</math>  <math>objs' = objs \cup \{o?\}</math>  <math>ocont' = ocont \oplus \{o? \mapsto \langle \rangle\}</math>  <math>osc' = osc \oplus \{o? \mapsto \theta SecClass\}</math>  <math>oacl' = oacl \oplus \{o? \mapsto \theta AccessCtrlList\}</math>  <math>aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}</math>  <math>repo! = ok</math> </div>
---	---

If preconditions are met, the system opens  $o?$  for writing for  $pid?$  (this is encoded in *POpenWrite*). Obviously,  $o?$ 's content is empty. In this case, the ACL resulting from *ACLSetOGO* is set as the initial ACL of the object as it is done in Linux. Also, the initial access class is set to  $L$ , i.e. the lowest access class.

The second case in "creating" an object is when the object exists. In this situation, the system must truncate the file and open it in *write* mode. Note that in this case  $perm?$  is not used, and also that we check for DAC access because  $(aprocs \ pid?).suid$  is not necessary one of the owners of the object (as is the case when the object does not exist). Given that no new name is added to  $parentDir \ o?$ , there is no need to check whether or not its access class dominates that of  $pid?$ .

*CreateOk2*

$\Delta FileSystemObjects$

$\Delta ProcessList$

$\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users$

*PreDACWrite*

*POpenWrite*

*pid?* : *PROCID*

*o?* : *OBJECT*

*perm?* : *OWNP*  $\times$  *GRPP*  $\times$  *OTHP*

*rep0!* : *SFSREPORT*

$pid? \in \text{dom } aprocs$

$o? \in objs$

$(aprocs \text{ } pid?) = \theta Process$

$ocont' = ocont \oplus \{o? \mapsto \langle \rangle\}$

$aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}$

$objs' = objs$

$oacl' = oacl$

$osc' = osc$

$rep0! = ok$

Error conditions are met when *pid?* does not exist or when the user on behalf of which *pid?* is acting does not have *write* permission for *o?* (second case, *E3*) or in *parentDir o?* (first case, *E1*).

$CreateE1 \triangleq MLSViolation$

$CreateE2 \triangleq PidNotExist$

$CreateE3 \triangleq NoWrite \wedge [\Xi SecureFileSystem; o? : OBJECT \mid o? \notin objs]$

$CreateE \triangleq CreateE1 \vee CreateE2 \vee CreateE3$

$CreateOk \triangleq CreateOk1 \vee CreateOk2$

$Create \triangleq CreateOk \vee CreateE$

**end of Z Section** *create*

#### 4.4.1 Design and implementation...

VER BIEN ESTO!!!!

We consider that a directory is empty when its state is equal to the state immediatly after it was created. Thus, this operation can be successffully invoked when a directory contains just '.' and '..'.

## 4.5 Exec

**Description** Executes a program

**Input parameters** *pid?* : *PROCID*; *o?* : *OBJECT*

**Kinds of objects** Programs, shell scripts

**Preconditions** The same of the standard Linux system call



**Postconditions** Basically the same of the standard Linux system call; the new process inherits the working access class of the caller process if it is not part of the TCB, otherwise the callee starts at  $L$

This operation represents the standard Linux system call **execve**. However, we have not modeled *execution* as a permission or mode. Instead, we consider *executing* as a kind of reading and so we check to see if  $pid?$  has *read* permission over  $o?$ . On the other hand, parameter  $o?$  represents both the executable and the libraries that must be loaded in order to execute it (and any other resource that is needed to execute the program).

## Z Section *exec*, parents: *state, definitions*

We start by establishing how a new process is built when it is executed by another process. Read the paragraph after schema *ExecOk* for an explanation.

$PExec$
$\Delta Process$
$usr' = usr$ $suid' = suid$ $lt' = lt$ $or' = or$ $ow' = ow$ $mmfr' = mmfr$ $mmfw' = mmfw$ $prog' = prog$ $mem' = \langle \rangle$

A given process may execute a program whenever the former has *read* permission over the last.

$ExecOk$
$\Delta ProcessList$ $\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi FileSystemObjects$ $PreDACRead$ $PExec$ $SCInit$ $pid? : PROCID$ $o? : OBJECT$ $repo! : SFSREPORT$
$pid? \in \text{dom } aprocs$ $(aprocs \text{ } pid?) = \theta Process$ $supr' =$ <b>if</b> $(aprocs \text{ } pid?).prog \in softtcb$ <b>then</b> $Sup \theta SecClass (osc \text{ } o?)$ <b>else</b> $Sup (aprocs \text{ } pid?).supr (osc \text{ } o?)$ $aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}$ $repo! = ok$

If the operation can proceed then, a new process is created with the same process identifier of the caller. The new process acts on behalf of the same user of  $pid?$ , and inherits the same open files of

$pid?$ . The memory of the new process is classified at  $L$  if the caller is part of the software portion of the TCB (*soft tcb*), otherwise it is classified at the working access class of  $pid?$ . The reason to classify the new process at the working access class of  $pid?$  when the last is not a trusted process, is because  $pid?$  may pass arguments containing classified information, or it could *Exec* a new process to signal some  $pid?$ 's condition to lower level users.

Errors are the standard ones: the requesting process does not exist, the program does not exist or the user does not have *read* permission over  $o?$ .

$$ExecE1 \triangleq PidNotExist$$

$$ExecE2 \triangleq ObjectNotExist$$

$$ExecE3 \triangleq NoRead$$

$$ExecE \triangleq ExecE1 \vee ExecE2 \vee ExecE3$$

$$Exec \triangleq ExecOk \vee ExecE$$

**end of Z Section *exec***

#### 4.5.1 Design and Implementation Comments

Trusted software is authorized to start processes at  $L$  because, precisely, it is trusted to do so in states that cannot affect the security of the system or because it launches other trusted programs. For example, program **login** is part of the TCB, and it is trusted to **exec** the user shell because we know that **login** is not a Trojan horse and so it will not signal anything to an untrusted program.

In saying this we warn programmers to carefully implement trusted programs.

### 4.6 Fork

**Description** Creates a new process

**Input parameters**  $pid? : PROCID$

**Preconditions** The same of the standard Linux system call

**Postconditions** Basically the same of the standard Linux system call; the new process inherits the working access class of the caller process

This operation represents the standard Linux system call **fork**. We suggest to read the design comments of operation *Exec*.

**Z Section *fork*, parents: *state, definitions***

A given process may fork itself at any time and without restrictions.

*ForkOk*

$\Delta ProcessList$

$\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi FileSystemObjects$

$pid? : PROCID$

$rep_0! : SFSREPORT$

$pid? \in \text{dom } aprocs$

$aprocs' = aprocs \oplus \{ \min\{p : PROCID \mid p \notin \text{dom } aprocs \bullet p\} \mapsto (aprocs \text{ } pid?) \}$

$rep_0! = ok$

The Linux semantics for `fork` is that the new process is exactly the same than its father. Then, the new process inherits the working access class of its father. The process identifier assigned to the forked process should be implemented as it is in Linux.

The only possible error is that the requesting process does not exist.

$$ForkE \triangleq PidNotExist$$

$$Fork \triangleq ForkOk \vee ForkE$$

**end of Z Section** *fork*

#### 4.6.1 Design and Implementation Comments

The manual page of `fork` says about the child process that “el uso de recursos esté asignado a 0”. It is necessary to investigate precisely what does it means.

### 4.7 Link

**Description** Creates a new name for an existing object

**Input parameters** *pid? : PROCID; old?, new? : OBJECT*

**Preconditions** The same of the `link` system call plus the access class of *o?*’s parent directory must dominate the access class of the process calling the function

**Postconditions** The same of the `link` system calls (i.e. security attributes remains the same in *new?*)

This operation represents the standard Linux system calls `link`. We have abstracted away many peculiarities of this system call. In fact, only its security features are captured by the following specification. Its remaining features must be implemented as they currently are in Linux.

**Z Section** *link, parents: state, definitions*

*new?* is linked to *old?*. In this case, *new?*’s attributes are those of *old?*, and *new?* is added to the set of existing objects.

*parentDir o?* may not belong to  $\text{dom } osc$  but we do not check this explicitly because this is a standard constrain in Linux. This does not mean that this checks should be removed from the implementation.

We require the access class of *new?*’s parent directory to dominate the access class of *pid?*’s memory in order to avoid illegal information flows in the form of file or directory names.

*LinkOk*

$\Delta \text{FileSystemObjects}$

$\Xi \text{ProcessList}; \Xi \text{UsersAndTerminals}; \Xi \text{LogicalTerminals}; \Xi \text{Users}$

$\text{pid?} : \text{PROCID}$

$\text{old?}, \text{new?} : \text{OBJECT}$

$\text{repo!} : \text{SFSREPORT}$

$\text{pid?} \in \text{dom } \text{aprocs}$

$\text{old?} \in \text{objs}$

$\text{new?} \notin \text{objs}$

$\text{osc } (\text{parentDir } \text{new?}) \succeq (\text{aprocs } \text{pid?}).\text{supr}$

$\text{objs}' = \text{objs} \cup \{\text{new?}\}$

$\text{ocont}' = \text{ocont} \oplus \{\text{new?} \mapsto (\text{ocont } \text{old?})\}$

$\text{osc}' = \text{osc} \oplus \{\text{new?} \mapsto \text{osc } \text{old?}\}$

$\text{oacl}' = \text{oacl} \oplus \{\text{new?} \mapsto (\text{oacl } \text{old?})\}$

$\text{repo!} = \text{ok}$

Error conditions are trivial.

$\text{LinkE1} \triangleq \text{ObjectNotExist}[\text{old?}/\text{o?}]$

$\text{LinkE2} \triangleq \text{ObjectAlreadyExists}[\text{new?}/\text{o?}]$

$\text{LinkE3} \triangleq \text{PidNotExist}$

$\text{LinkE4} \triangleq \text{MLSViolation}[\text{new?}/\text{o?}]$

$\text{LinkE} \triangleq \text{LinkE1} \vee \text{LinkE2} \vee \text{LinkE3} \vee \text{LinkE4}$

$\text{Link} \triangleq \text{LinkOk} \vee \text{LinkE}$

end of Z Section *link*

## 4.8 LinkS

**Description** Creates a new name for an existing object

**Input parameters**  $\text{pid?} : \text{PROCID}; \text{old?}, \text{new?} : \text{OBJECT}$

**Preconditions** The same of the `symlink` system call plus the access class of  $\text{o?}$ 's parent directory must dominate the access class of the process calling the function

**Postconditions** The same of the `symlink` system calls but the access class of  $\text{new?}$  equals the access class of the calling process.

This operation represents the standard Linux system calls `symlink`. We have abstracted away many peculiarities of this system call. In fact, only its security features are captured by the following specification. Its remaining features must be implemented as they currently are in Linux.

## Z Section *links, parents: state, definitions*

*new?* is linked to *old?*. The last object may not exist in the current file system, but the former must not exist.

*parentDir o?* may not belong to  $\text{dom } osc$  but we do not check this explicitly because this is a standard constrain in Linux. This does not mean that this checks should be removed from the implementation.

We require the access class of *new?*'s parent directory to dominate the access class of *pid?*'s memory in order to avoid illegal information flows in the form of file or directory names.

The content of *new?* is the name of *old?*. This is a fundamental difference with respect to *Link* specification (see section 4.7), because this fact implies that *LinkS* performs a *Write* into *new?*. Given that the type of variable *new?* and *ocont* are not compatible we need to define a function mapping *OBJECT* onto sequences of *CHAR*. In this way we are able to add to *ocont* the content of *new?*.

|  $OBJECTToSeqCHAR : OBJECT \rightarrow \text{seq } CHAR$

*LinkSOk*

$\Delta FileSystemObjects$

$\Xi ProcessList; \Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users$

*pid?* : *PROCID*

*old?, new?* : *OBJECT*

*rep<sub>0</sub>!* : *SFSREPORT*

*pid?*  $\in \text{dom } aprocs$

*new?*  $\notin objs$

*osc* (*parentDir new?*)  $\succeq (aprocs \text{ } pid?).supr$

*objs'* = *objs*  $\cup \{new?\}$

let *NEWCONT* ==  $(\lambda i : 1 \dots \#(OBJECTToSeqCHAR \text{ } old?) \bullet$   
 $((OBJECTToSeqCHAR \text{ } old?)i, (aprocs \text{ } pid?).supr)) \bullet$

*ocont'* = *ocont*  $\oplus \{new? \mapsto NEWCONT\}$

*osc'* = *osc*  $\oplus \{new? \mapsto (aprocs \text{ } pid?).supr\}$

*oacl'* = *oacl*  $\oplus \{new? \mapsto (oacl \text{ } old?)\}$

*rep<sub>0</sub>!* = *ok*

If preconditions are met we set the following postconditions:

- *new?* is added to the set of objects
- The content of *new?* is set to the set of characters constituting th name of *old?*, and each of these characters are classified with the access class of the calling process. This is so, because the access class of *new?* is set to the same access class (see next item)
- The access class of *new?* is set to the access class of the calling process. We cannot set it to *L* because this operation takes characters from the process' memory space and writes them into the file. Then, if the process is classified above *L* setting the access class of *new?* to *L* allows an illegal information flow.
- *new?*'s DAC attributes are those of *old?*

Error conditions are trivial.

$$LinkSE1 \triangleq ObjectAlreadyExists[new?/o?]$$

$$LinkSE2 \triangleq PidNotExist$$

$$LinkSE3 \triangleq MLSEViolation[new?/o?]$$

$$LinkSE \triangleq LinkSE1 \vee LinkSE2 \vee LinkSE3$$

$$LinkS \triangleq LinkSOk \vee LinkSE$$

end of Z Section *links*

## 4.9 Mmap

**Description** Maps a file into memory

**Input parameters**  $pid? : PROCID$ ;  $o? : OBJECT$ ;  $m? : MODE$

**Preconditions**  $o?$  had to be opened in a mode not in conflict with  $m?$ ; if the file is being mapped in *read* mode then its access class must be dominated by the access class of the greatest lower bound of the files already mapped in *write* mode

**Postconditions**  $pid?$  can access  $o?$  directly from its memory space, i.e. it is not necessary to use *Read* or *Write* (see sections 4.12 and 4.17)

This operation represents the standard Linux system call `mmap` (see its manual page for more details). `mmap` is very important to security because it allows processes to access files without using `read` or `write`. In fact, if a process has mapped a file into memory then the process can access the file like a memory buffer, i.e. without calling the kernel.

The system call has a parameter, called `flags` in its manual page, that has not been included in our model in order to keep it simple. This parameter in conjunction with  $m?$  has security implications. For example, if `flags` is set to `MAP_PRIVATE` and  $m?$  to *write*, all modifications to the mapped file are private to the process. This means that modifications are not saved when the file is unmapped.

Thus, to keep the model simple we decided that to map a file in *write* mode means that modifications to the mapped file are saved when it is unmapped, otherwise the file has been mapped in *read* mode. In other words, if the model specifies a case where a file is mapped in *write* mode it represents, at implementation level, an invocation of `mmap` where the combination between `prot` and `flags` allows changes to be saved.

**Z Section *mmap*, parents:** *state, definitions*

First, we define a frame schema so then we promote the operation to the system level.

$PMmap$
$\Delta Process$
$usr' = usr$
$suid' = suid$
$lt' = lt$
$or' = or$
$ow' = ow$
$prog' = prog$

Then, we define two schemas one for each mode in which a file can be mapped. In any case, a file can be mapped in a given mode if it was previously opened in the same mode. The first schema represents the case when a file is mapped in *read* mode; here we need an extra precondition:

$$INF\{o : OBJECT \mid o \in (aprocs \text{ } pid?).mmfw \bullet osc \ o\} \succeq osc \ o?$$

to prevent the situation where a process maps a file in *read* mode but it already has another, lower level file mapped in *write* mode. In that case the process would be able to read from a higher file and to copy its contents in a lower level file. This precondition looks like the precondition needed to preserve confinement in BLP-like models [2, 3].

$MmapOk1$ $\Delta ProcessList$ $\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi FileSystemObjects$ $PMmap$ $pid? : PROCID$ $o? : OBJECT$ $m? : MODE$ $rep0! : SFSREPORT$
$pid? \in \text{dom } aprocs$ $o? \in objs$ $m? = read$ $o? \in or$ $INF\{o : OBJECT \mid o \in (aprocs \text{ } pid?).mmfw \bullet osc \ o\} \succeq osc \ o?$ $aprocs \text{ } pid? = \theta Process$ $mmfr' = mmfr \cup \{o?\}$ $supr' = Sup \ supr \ (osc \ o?)$ $mem' = mem \wedge put\_in\_mem(ocont \ o?)$ $mmfw' = mmfw$ $aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}$ $rep0! = ok$

If preconditions are met then some part of the file is copied into the memory space of  $pid?$ <sup>1</sup>. Hence,  $supr$  must be updated accordingly; that is it must be set to the least upper bound between its actual value and the access class of  $o?$ .

Schema  $MmapOk2$  represents the case of mapping a file in *write* mode<sup>2</sup>. This is possible only if the file was previously opened in the same mode. Given that a mapped file is accessible without the intervention of the kernel, we must forbid the operation if the access class of the file to be mapped is not dominated by the working access class of  $pid?$ . This is so because otherwise  $pid?$  could write information from its memory into  $o?$  without the kernel controlling it.

<sup>1</sup>Obviously this particular feature must be implemented as it is in Linux. Further, we do not model the fact that this memory pages are marked as read-only making it impossible to write on them.

<sup>2</sup>Again, we have not modeled the marking of memory pages, thus the process is able to read this pages. Obviously, this must be implemented as it is in Linux.

$MmapOk2$ $\Delta ProcessList$ $\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi FileSystemObjects$ $PMmap$ $pid? : PROCID$ $o? : OBJECT$ $m? : MODE$ $repo! : SFSREPORT$
$pid? \in \text{dom } aprocs$ $o? \in objs$ $m? = write$ $o? \in ow$ $osc\ o? \succeq (aprocs\ pid?).supr$ $(aprocs\ pid?) = \theta Process$ $mmfw' = mmfw \cup \{o?\}$ $mem' = mem \wedge put\_in\_mem(ocont\ o?)$ $supr' = supr$ $mmfr' = mmfr$ $aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}$ $repo! = ok$

It is an standard error to try to map a file in mode  $m?$  if it has not been opened in that mode. We have added one more error if the file is mapped in *write* mode (this was explained in the previous paragraph). The remaining errors are easy to understand.

$MmapE1$ $\Xi SecureFileSystem$ $pid? : PROCID$ $o? : OBJECT$ $m? : MODE$ $repo! : SFSREPORT$
$pid? \in \text{dom } aprocs$ $o? \in objs$ $(m? = read \wedge o? \notin (aprocs\ pid?).or$ $\vee m? = write \wedge (o? \notin (aprocs\ pid?).ow \vee \neg osc\ o? \succeq (aprocs\ pid?).supr))$ $repo! = permissionDenied$

$$MmapE2 \triangleq PidNotExist$$

$$MmapE3 \triangleq ObjectNotExist$$

$$MmapE \triangleq MmapE1 \vee MmapE2 \vee MmapE3$$

$$MmapOk \triangleq MmapOk1 \vee MmapOk2$$

$$Mmap \triangleq MmapOk \vee MmapE$$



end of Z Section *mmap*

## 4.10 Open

**Description** Opens an object in a given mode

**Input parameters**  $pid? : PROCID$ ;  $o? : OBJECT$ ;  $m? : MODE$

**Kinds of objects** It depends on what particular system call *Open* represents. If it is **open**, then  $o?$  is a file; if it is **opendir**, then  $o?$  is a directory

**Preconditions** The standard DAC checks

**Postconditions**  $o?$  is added to the list of open files of  $pid?$

This operation represents the standard **open** system call of Linux. If a process wants to read or write a file it first needs to open the file. The system call returns a file descriptor which is used by the process for future references to the file; this was not included in our model.

This description must be used as a guide to implement similar system calls like **opendir**. See [6] for more details.

**Z Section** *open*, **parents:** *state, definitions*

The following schema is used to promote the operation from the process level to the system level.

$OpenFrame$ $\Delta ProcessList$ $\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi FileSystemObjects$ $\Delta Process$ $pid? : PROCID$ $o? : OBJECT$ $m? : MODE$ $rep_0! : SFSREPORT$ <hr/> $aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}$ $rep_0! = ok$
--

Finally, the operation is defined by cojoining the frame with the schemas at the process level and the schemas recording the standard DAC preconditions. Note that we do not require any MLS controls as is suggested in [2, 3]. Instead, we enforce an information flow policy similar to that presented in [7], see sections 4.12 and 4.17.

$OpenOk1 \triangleq [OpenFrame; PreDACRead; POpenRead \mid m? = read]$

$OpenOk2 \triangleq [OpenFrame; PreDACWrite; POpenWrite \mid m? = write]$

$OpenOk \triangleq OpenOk1 \vee OpenOk2$

Error schemas capture all possible errors.

$OpenE1 \triangleq PidNotExist$

$OpenE2 \triangleq ObjectNotExist$

$OpenE31 \triangleq NoRead$

$OpenE32 \triangleq NoWrite$

$OpenE3 \triangleq OpenE31 \vee OpenE32$

$OpenE \triangleq OpenE1 \vee OpenE2 \vee OpenE3$

$Open \triangleq OpenOk \vee OpenE$

**end of Z Section** *open*

## 4.11 Oscstat

**Description** Returns the access class of a given object

**Input parameters**  $pid? : PROCID$ ;  $o? : OBJECT$

**Preconditions** None

**Postconditions**  $o?$ 's access class is copied to  $pid?$ 's memory

This operation represents system call `oscstat`. It drastically changes the semantics of this call with respect to Lisex. Now, any process may request the access class of any object.

**Z Section** *oscstat*, **parents:** *state, definitions*

We need to copy a security class into a process' memory space. Processes' memory spaces stores a sequence of *CCHAR* which is a different type than *SecClass*. So, we define a function mapping *SecClass* onto seq *CHAR* (then, in schema *OscstatOk*, we will transform a sequence of *CHAR* into a sequence of *CCHAR*).

|  $SCToSeqCHAR : SecClass \rightarrow \text{seq } CHAR$

We use a lambda expression to define what is copied to  $pid?$ 's memory. Note that, by not updating  $(aprocs\ pid?).supr$ , we decided to classify this information at *L*. Future versions of GTL may classify it at  $o?$ 's access class.

*OscstatOk*

$\Delta ProcessList$

$\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi FileSystemObjects$

*PAddToMem*

*SCInit*

$pid? : PROCID$

$o? : OBJECT$

$repo! : SFSREPORT$

$pid? \in \text{dom } aprocs$

$o? \in objs$

**let**  $sc == SCToSeqCHAR (osc\ o?) \bullet$

$\quad buff = (\lambda i : 1.. \#sc \bullet (sc\ i, \theta SecClass))$

$aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}$

$repo! = ok$

Errors are quite simple to deserve further comments.

$OscstatE1 \triangleq PidNotExist$

$OscstatE2 \triangleq ObjectNotExist$

$OscstatE \triangleq OscstatE1 \vee OscstatE2$

$Oscstat \triangleq OscstatOk \vee OscstatE$

end of Z Section *oscstat*

## 4.12 Read

**Description** Reads from an open object

**Input parameters** *pid?* : *PROCID*; *o?* : *OBJECT*

**Kinds of objects** It depends on what particular system call *Read* represents. If it is **read**, then *o?* is a file; if it is **readdir** or **getdents**, then *o?* is a directory

**Preconditions** The object must be opened in *read* mode by *pid?*

**Postconditions** All of the characters read from *o?* are copied in *pid?*'s memory space

This operation represents the standard **read** system call. It should be used as a guide for the implementation of similar system calls like **readdir** or **getdents**. See [6] for more details.

The correct implementation of this operation is tantamount to the security of the system because it records some state data which is latter used by *Write* to decide if an object may be written by a process.

The specification we introduce is a convenient abstraction of the system call. We have omitted the following two parameters:

**buf** where in memory must bytes be copied, and

**nbytes** how many bytes must be read.

Instead, we take **nbytes** as the lengh of the file, and **buf** as the next memory address following the last being used (i.e. we add the entire file at the end of the process memory space). Obviously, this abstraction must be implemented as currently is in Linux.

The reason to abstract the system call in this way is based on the fact that we classify the entire memory space at the same access class, so it is unimporant where bytes are copied. Moreover, we do not care how many bytes are read because if the process read just one byte its memory space must be re-classified.

**Z Section** *read*, **parents:** *state, definitions*

Schema *PRead* is used to set the process state after a *Read* operation. A *Read* operation can proceed if *pid?* is an existent process, *o?* is opened in *read* mode by *pid?*, and when there is no memory mapped files in *write* mode with access classes lower than that of *o?*.

$ReadOk$ $\Delta ProcessList$ $\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi FileSystemObjects$ $PRead$ $pid? : PROCID$ $o? : OBJECT$ $repo! : SFSREPORT$
$pid? \in \text{dom } aprocs$ $o? \in (aprocs \text{ } pid?).or$ $INF\{o : OBJECT \mid o \in (aprocs \text{ } pid?).mmfw \bullet osc \ o\} \succeq osc \ o?$ $sc = (osc \ o?) \wedge buff = read\_inode(ocont \ o?)$ $(aprocs \text{ } pid?) = \theta Process$ $aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}$ $repo! = ok$

If preconditions are met, we record the fact that  $pid?$  has read  $o?$  by updating  $supr$ , and by copying part of the file into  $pid?$ 's memory. These predicates are hidden in  $PRead$ :  $supr' = Sup \ supr \ sc$ , where  $Sup$  is the binary upper bound operator on  $SecClass$ ; and  $mem' = mem \wedge buff$  where  $buff$  equals  $(ocont \ o?)$ . The first predicate states that the working access class of  $pid?$  is updated to the least upper bound between the current working access class of  $pid?$  and the access class of  $o?$ .

It is an error to request a *Read* operation over an object not open in *read* mode, and if  $pid?$  is not a process. Also, an error condition must be returned if there are lower files mapped on memory in *write* mode.

$ReadE1$ $\Xi SecureFileSystem$ $pid? : PROCID$ $o? : OBJECT$ $repo! : SFSREPORT$
$pid? \in \text{dom } aprocs$ $o? \notin (aprocs \text{ } pid?).or$ $repo! = objectIsNotOpenForReading$

$ReadE2$ $\Xi SecureFileSystem$ $pid? : PROCID$ $o? : OBJECT$ $repo! : SFSREPORT$
$pid? \in \text{dom } aprocs$ $o? \in objs$ $\neg INF\{o : OBJECT \mid o \in (aprocs \text{ } pid?).mmfw \bullet osc \ o\} \succeq osc \ o?$ $repo! = permissionDenied$

$ReadE3 \triangleq PidNotExist$

$ReadE \triangleq ReadE1 \vee ReadE2 \vee ReadE3$

$Read \triangleq ReadOk \vee ReadE$

**end of Z Section** *read*

#### 4.12.1 Design and Implementation Comments

In Linux, when `nbytes` are read from a file, the operating system kernel increments an internal pointer so that successive reads start from position `nbytes + 1`. We have not modeled this feature because it adds nothing to security.

On the other hand, it is not always necessary to recalculate *supr* as schema *PRead* suggests. It is only necessary when a file is read for first time or when its access class was changed since the first read. Then, if the kernel internally records whether or not a file has been read, postcondition  $supr' = Sup\ supr' \ sc$  should only be executed for files that have not been read (or when its access class has changed since the last read)<sup>3</sup>. See section 4.17 for further comments.

The same happens with precondition

$$INF\{o : OBJECT \mid o \in mmfw \bullet osc\ o\} \succeq osc\ o?$$

That is, at implementation level, this predicate can be checked just when the file is read for the first time. This is so, because if this predicate is true the first time the file is read, then, given the specification for *Mmap*, it will be impossible to map lower files in *write* mode.

This operation could be implemented jointly with *ReadLT*.

### 4.13 ReadLT

**Description** Reads from a logical terminal

**Input parameters** *pid?* : *PROCID*

**Preconditions** *pid?* must be a valid process

**Postconditions** The characters read from the logical terminal which *pid?* is connected to are copied in *pid?*'s memory space

This operation is the **read** system call when the object to read is a logical terminal connected to the process issuing the call. It may be implemented as part of the code of **read**. We have modeled it as different of *Read* because in our model logical terminals have a different type (and different properties) than objects. Also, we model in *ReadLT* how characters in *stdin* are consumed as they are read.

We strongly recommend to read *Read* and *Login* descriptions before implementing this operation. Regarding *Read*, similar design considerations apply to this operation.

**Z Section** *readlt*, **parents:** *state*, *definitions*

This schema will be promoted to the system level. It simply says that a *ReadLT* operation consumes the input available in the logical terminal.

<i>LTCRead</i>	_____
$\Delta LTCont$	
<i>stdin'</i> = $\langle \rangle$	
<i>stdout'</i> = <i>stdout</i>	

Information can flow from a terminal to a process only if the process has no lower memory mapped files in *write* mode (see section 4.9 for more details). This is stated in the first **let** construct.

<sup>3</sup>Thanks to Alejandro Hernández and Felipe Manzano for noticing part of this optimization.

The last two **let** constructs define the logical terminal identifier of the logical terminal which  $pid?$  is connected to. We have used two, instead of just one, to separate predicates on variables of *ProcessList* from those of *LogicalTerminals*. By cojoining the predicates of the second **let** with those predicates defined in *PRead*, we define how the process is updated after reading from its terminal. Similarly, *LTRed* and the third **let** state how the logical terminal changes its state.

$ReadLTok$
$\Delta ProcessList$ $\Delta LogicalTerminals$ $\Xi UsersAndTerminals; \Xi Users; \Xi FileSystemObjects$ $PRead$ $LTCRead$ $pid? : PROCID$ $repo! : SFSREPORT$
$pid? \in \text{dom } aprocs$ <b>let</b> $p == (aprocs \text{ } pid?) \bullet$ $INF\{o : OBJECT \mid o \in p.mmfw \bullet osc \ o\} \succeq ltsc \ p.lt$ <b>let</b> $lt == (aprocs \text{ } pid?).lt \bullet$ $sc = ltsc \ lt$ $\wedge buff = (ltcont \ lt).stdin$ $\wedge aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}$ <b>let</b> $lt == (aprocs \text{ } pid?).lt \bullet$ $ltcont' = ltcont \oplus \{lt \mapsto \theta LTCont'\}$ $repo! = ok$

An error condition must be returned if there are lower files mapped on memory in *write* mode.

$ReadLTE1$
$\Xi SecureFileSystem$ $pid? : PROCID$ $repo! : SFSREPORT$
$pid? \in \text{dom } aprocs$ <b>let</b> $p == (aprocs \text{ } pid?) \bullet$ $\neg INF\{o : OBJECT \mid o \in p.mmfw \bullet osc \ o\} \succeq ltsc \ p.lt$ $repo! = permissionDenied$

$ReadLTE2 \triangleq PidNotExist$

$ReadLTE \triangleq ReadLTE1 \vee ReadLTE2$

$ReadLT \triangleq ReadLTok \vee ReadLTE$

**end of Z Section** *readlt*

## 4.14 Rename

**Description** Changes the name and/or the path of a file or directory

**Input parameters**  $pid? : PROCID; old?, new? : OBJECT$

**Preconditions** The same of the standard Linux system call plus the access class of  $new?$ 's parent directory must dominates the access class of the process calling the function

**Postconditions** The same of the standard Linux system call (i.e. security attributes remains the same in  $new?$ )

This operation represents the standard Linux system call **rename**.

## Z Section *rename*, **parents**: *state, definitions*

$parentDir\ o?$  may not belong to  $\text{dom } osc$  but we do not check this explicitly because this is a standard constrain in Linux. This does not mean that this checks should be removed from the implementation.

We require the access class of  $new?$ 's parent directory to dominate the access class of  $pid?$ 's memory in order to avoid illegal information flows in the form of file or directory names.

*RenameOk*

$\Delta \text{FileSystemObjects}$

$\Xi \text{ProcessList}; \Xi \text{UsersAndTerminals}; \Xi \text{LogicalTerminals}; \Xi \text{Users}$

$pid? : \text{PROCID}$

$old?, new? : \text{OBJECT}$

$repo! : \text{SFSREPORT}$

$pid? \in \text{dom } aprocs$

$old? \in \text{objs}$

$new? \notin \text{objs}$

$osc\ (parentDir\ new?) \succeq (aprocs\ pid?).supr$

$objs' = (objs \setminus \{old?\}) \cup \{new?\}$

$ocont' = (ocont \setminus \{old? \mapsto ocont\ old?\}) \cup \{new? \mapsto ocont\ old?\}$

$osc' = (osc \setminus \{old? \mapsto osc\ old?\}) \cup \{new? \mapsto osc\ old?\}$

$oacl' = (oacl \setminus \{old? \mapsto oacl\ old?\}) \cup \{new? \mapsto oacl\ old?\}$

$repo! = ok$

All the functional variables in *FileSystemObjects* must be updated in the same manner: a pair of the form  $old? \mapsto f\ old?$  has to be removed from  $f$ , while a pair of the form  $new? \mapsto f\ old?$  is added to function  $f$ .

Error conditions are trivial.

$RenameE1 \triangleq \text{ObjectNotExist}[old?/o?]$

$RenameE2 \triangleq \text{ObjectAlreadyExists}[new?/o?]$

$RenameE3 \triangleq \text{PidNotExist}$

$RenameE4 \triangleq \text{MLSViolation}[new?/o?]$

$RenameE \triangleq RenameE1 \vee RenameE2 \vee RenameE3$

$Rename \triangleq RenameOk \vee RenameE$

end of Z Section *rename*

## 4.15 Setuid

**Description** Sets the user identity of a process

**Input parameters** *pid?* : *PROCID*; *new?* : *USER*

**Preconditions** *root* can change to a MAC administrator only when the process requesting *Setuid* is a trusted process; *root* can change to any user (except to *secadm*) in any other circumstances; ordinary users can change to the (Linux) owner of the program (unless the owner is a MAC administrator) when its SUID bit is on

**Postconditions** *pid?* starts to act on behalf of *new?*

This operation represents the family of system calls based on `suid`. We vaguely model the fact that a process acting on behalf of a user different than *root* can successfully request *Setuid* if the SUID bit is on in the program that originated the process. This functionality must be implemented as it is in Linux.

**Z Section** *setuid*, **parents:** *state, definitions*

We start with a framing schema to be used latter to promote the main operation. This schema unconditionally changes the user to a new user received as input.

<i>PSetuid</i>
$\Delta Process$
<i>new?</i> : <i>USER</i>
$suid' = new?$
$usr' = usr$
$lt' = lt$
$or' = or$
$ow' = ow$
$mmfr' = mmfr$
$mmfw' = mmfw$
$supr' = supr$
$prog' = prog$
$mem' = mem$

An untrusted process may request *Suid* if it is acting on behalf of *root* and the new user is not a MAC administrator. We test whether *new?* is a MAC administrator by checking if *SECADMIN* is a category in *new?*'s security class.

We forbid *root* to set the identity of its untrusted processes to a MAC administrator, because otherwise a Trojan horse running on behalf of *root* could change its identity to a MAC administrator and then it would be able to successfully invoke critical operations such as *Chobjsc*. This kind of attacks indirectly affect confidentiality.



$SetuidOk1$ $\Delta ProcessList$ $\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi FileSystemObjects$ $\Delta Process$ $PSetuid$ $pid? : PROCID$ $new? : USER$ $rep0! : SFSREPORT$
$pid? \in \text{dom } aprocs$ $new? \in users$ $(aprocs \text{ } pid?).usr = root$ $SECADMIN \notin (usc \text{ } new?).categs$ $(aprocs \text{ } pid?).prog \notin softtcb$ $(aprocs \text{ } pid?) = \theta Process$ $aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}$ $rep0! = ok$

See how operation promotion is used:

1. A framing schema is defined ( $PSetuid$ )
2. Preconditions are set in the operation schema ( $PSetuidOk1$ )
3. In particular, we set  $aprocs \text{ } pid? = \theta Process$ , so any unprimed variable defined in the framing schema ( $PSetuid$ ) equals the same variable of the interesting process. For example variable  $usr$  in  $PSetuid$  equals  $(aprocs \text{ } pid?).usr$ .
4. Also, by setting  $(aprocs \text{ } pid?).usr = root$  we indirectly set  $usr = root$  in  $PSetuid$
5. Finally, postconditions are set by using  $\theta Process'$ ; that is, the new process associated with  $pid?$  has all its state variables with the same value as the old process except  $user$  which equals  $new?$ .

If the process invoking  $Setuid$  is acting on behalf of  $root$  and is the result of executing a trusted program then it is authorized to change its identity to a MAC administrator.

$SetuidOk2$ $\Delta ProcessList$ $\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi FileSystemObjects$ $\Delta Process$ $PSetuid$ $pid? : PROCID$ $new? : USER$ $rep0! : SFSREPORT$
$pid? \in \text{dom } aprocs$ $new? \in users$ $(aprocs \text{ } pid?).usr = root$ $SECADMIN \in (usc \text{ } new?).categs$ $(aprocs \text{ } pid?).prog \in softtcb$ $(aprocs \text{ } pid?) = \theta Process$ $aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}$ $rep0! = ok$

Finally, if a process acting on behalf of a regular user invokes *Setuid* then the system sets the identity of the process to the user indicated by *suidto*.

$\text{SetuidOk3}$ $\Delta \text{ProcessList}$ $\Xi \text{UsersAndTerminals}; \Xi \text{LogicalTerminals}; \Xi \text{Users}; \Xi \text{FileSystemObjects}$ $\Delta \text{Process}$ $P\text{Setuid}$ $\text{pid?} : \text{PROCID}$ $\text{new?} : \text{USER}$ $\text{repo!} : \text{SFSREPORT}$
$\text{pid?} \in \text{dom } \text{aprocs}$ $\text{new?} \in \text{users}$ $(\text{aprocs } \text{pid?}).\text{usr} \neq \text{root}$ $\text{SECADMIN} \notin (\text{usc } \text{new?}).\text{categs}$ $\text{new?} = \text{suidto } (\text{aprocs } \text{pid?}).\text{prog}$ $(\text{aprocs } \text{pid?}) = \theta \text{Process}$ $\text{aprocs}' = \text{aprocs} \oplus \{\text{pid?} \mapsto \theta \text{Process}'\}$ $\text{repo!} = \text{ok}$

Errors are the standard ones. The first one captures all the possible situations where the process lacks the necessary permissions to invoke the operation.

$\text{SetuidE1}$ $\Xi \text{SecureFileSystem}$ $\text{pid?} : \text{PROCID}$ $\text{new?} : \text{USER}$ $\text{repo!} : \text{SFSREPORT}$
$\text{pid?} \in \text{dom } \text{aprocs}$ $\text{new?} \in \text{users}$ $((\text{aprocs } \text{pid?}).\text{usr} = \text{root})$ $\quad \wedge \text{SECADMIN} \in (\text{uscnew?}).\text{categs}$ $\quad \wedge (\text{aprocs } \text{pid?}).\text{prog} \notin \text{softtcb}$ $\vee (\text{aprocs } \text{pid?}).\text{usr} \neq \text{root}$ $\quad \wedge \text{SECADMIN} \in (\text{uscnew?}).\text{categs}$ $\vee (\text{aprocs } \text{pid?}).\text{usr} \neq \text{root}$ $\quad \wedge \text{new?} \neq \text{suidto } (\text{aprocs } \text{pid?}).\text{prog}$ $\text{repo!} = \text{permissionDenied}$

$\text{SetuidE2} \triangleq \text{PidNotExist}$

$\text{SetuidE3} \triangleq \text{UserNotExist}$

$\text{SetuidE} \triangleq \text{SetuidE1} \vee \text{SetuidE2} \vee \text{SetuidE3}$

$\text{SetuidOk} \triangleq \text{SetuidOk1} \vee \text{SetuidOk2} \vee \text{SetuidOk3}$

$\text{Setuid} \triangleq \text{SetuidOk} \vee \text{SetuidE}$

end of Z Section *setuid*

## 4.16 Stat

**Description** Returns state information of a given object (not including MAC attributes)

**Input parameters**  $pid? : PROCID$ ;  $o? : OBJECT$

**Preconditions** The user issuing the call or some group to which the user belongs must be in some entry of  $o?$ 's ACL

**Postconditions**  $o?$ 's ACL is copied to  $pid?$ 's memory

This operation represent various system calls such as **stat**, **aclstat**, and so on. It drastically changes the semantics of these calls with respect to Linux. Now, a user does not need **x** mode along the path to  $o?$  to stat it, instead he or she needs to be part of  $o?$ 's ACL.

**Z Section** *stat*, **parents:** *state*, *definitions*

We need to copy an ACL into a process' memory space. Processes' memory spaces stores a sequence of *CCHAR* which is a different type than *AccessCtrlList*. So, we define a function mapping *AccessCtrlList* onto seq *CHAR* (then we will transform a sequence of *CHAR* into a sequence of *CCHAR*).

|  $ACLToSeqCHAR : AccessCtrlList \rightarrow \text{seq } CHAR$

Expand schema *PreDAC* to see what preconditions are required. There you will see that *suid*, and not *usr*, is used to check for DAC permissions.

*StatOk*

$\Delta ProcessList$

$\Xi UsersAndTerminals$ ;  $\Xi LogicalTerminals$ ;  $\Xi Users$ ;  $\Xi FileSystemObjects$

*PreDAC*

*PAddToMem*

*SCInit*

$pid? : PROCID$

$o? : OBJECT$

$rep_0! : SFSREPORT$

**let**  $acl == ACLToSeqCHAR (oacl\ o?) \bullet$   
 $buff = (\lambda i : 1 \dots \#acl \bullet (acl\ i, \theta SecClass))$   
 $aprocs' = aprocs \oplus \{pid? \mapsto \theta Process'\}$   
 $rep_0! = ok$

Note that the information added to  $pid?$ 's memory is classified at *L* (and so we do not need to update *supr* because  $sc = Sup\ sc\ L$  for every security class *sc*).

We only model how this operation returns the ACL of the object: at implementation level, system calls must return the information originally intended. For example, **stat** must return a **stat** struct, **aclstat** must return just the ACL, and so on.

On the other hand, to model this operation we directly access the secret of *AccessCtrlList*, but at implementation level *AccessCtrlList*'s interface must be used.

Errors are the obvious:  $pid?$  or  $o?$  may not exist, and the user requesting the action may lack the necessary rights (that is  $pid?$ 's *suid* is not part of  $o?$ 's ACL).

$$\begin{aligned}
StatE1 &\triangleq PidNotExist \\
StatE2 &\triangleq ObjectNotExist \\
StatE3 &\triangleq (NoRead \wedge NoWrite \wedge NoOwner) \setminus (m?) \\
StatE &\triangleq StatE1 \vee StatE2 \vee StatE3 \\
Stat &\triangleq StatOk \vee StatE
\end{aligned}$$

**end of Z Section** *stat*

## 4.17 Write

**Description** Writes to an open object

**Input parameters** *pid?* : *PROCID*; *o?* : *OBJECT*

**Kinds of objects** Files

**Preconditions** The object must be opened in *write* mode by *pid?*

**Postconditions** The characters written by *pid?* are copied into *o?*'s content

This operation represents the standard **write** system call. Its specification should be used to program all write functions declared to VFS<sup>4</sup>, except the version used to write into ttys which is specified in section 4.18.

The correct implementation of this operation is tantamount to the security of the system because it forbids the downgrade of information. Also, we have introduced an enhancement that may increase the usability of the system.

The specification we introduce is a convenient abstraction of the system call. We have omitted the following two parameters:

**buff** to indicate where in memory are the bytes to be written.

**num** to indicate how many bytes must be written, and

We decided not to model these features because they do not add anything to the problem of security.

In what follows, note that **root** has no special privileges when requesting this operation. That is to say, **root** cannot violate *Write*. More generally, **root** has no special privileges with respect to the MLS model.

**Z Section** *write*, **parents:** *state*, *definitions*

There are two successful cases depending on whether the file to be written is empty or not. *WriteOk1* is the case to be used when *o?* is not empty. In this case, the system performs the operation only if the working access class of *pid?* is dominated by the access class of *o?*. The reason to enforce this restriction is clear: a process writes a file with data taken from its memory space which in turn contains data read from several sources; thus, if one of these sources is highly classified then the process may disclose information if this check is not performed.

The exact way the system writes into a file depends on previous writes and how the process had moved the read/write pointer of the object. We have encoded this features in *write\_inode*, which is underspecified.

---

<sup>4</sup>We mean the **write\_inode** field of the **inode\_operations** structure.

---

*WriteOk1*

---

$\Delta FileSystemObjects$

$\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi ProcessList$

$pid? : PROCID$

$o? : OBJECT$

$rep_0! : SFSREPORT$

---

$pid? \in \text{dom } aprocs$

$o? \in (aprocs \text{ } pid?).ow$

$ocont \text{ } o? \neq \langle \rangle$

$osc \text{ } o? \succeq (aprocs \text{ } pid?).supr$

$ocont' = ocont \oplus \{o? \mapsto write\_inode (ocont \text{ } o?) (aprocs \text{ } pid?).mem\}$

$objs' = objs$

$oacl' = oacl$

$osc' = osc$

$rep_0! = ok$

---

Note that with this semantics it is possible that two *Writes* on the same file may have different behavior if in the mean time the process read from a file with an access class dominating its working access class.

When  $o?$  is empty, the system does not check the working access class of  $pid?$  against that of  $o?$ . In this case, the process is authorized to write anything into the object but the object's access class is set to the working access class of the process.

---

*WriteOk2*

---

$\Delta FileSystemObjects$

$\Xi UsersAndTerminals; \Xi LogicalTerminals; \Xi Users; \Xi ProcessList$

$pid? : PROCID$

$o? : OBJECT$

$rep_0! : SFSREPORT$

---

$pid? \in \text{dom } aprocs$

$o? \in (aprocs \text{ } pid?).ow$

$ocont \text{ } o? = \langle \rangle$

$ocont' = ocont \oplus \{o? \mapsto write\_inode (ocont \text{ } o?) (aprocs \text{ } pid?).mem\}$

$osc' = osc \oplus \{o? \mapsto (aprocs \text{ } pid?).supr\}$

$objs' = objs$

$oacl' = oacl$

$rep_0! = ok$

---

We believe that this semantics will increase the usability of the system because objects such as *pipes* will be able to receive information from many sources provided they are emptied before a *Write* is requested.

It is an error to try to write into a non open object.

$WriteE1$ $\exists SecureFileSystem$ $pid? : PROCID$ $o? : OBJECT$ $rep_0! : SFSREPORT$
$pid? \in \text{dom } aprocs$ $o? \notin (aprocs \text{ } pid?).ow$ $rep_0! = objectIsNotOpenForWriting$

An error is returned when a process requests a write into a non empty object while it is working at an access class that dominates that of the object.

$WriteE2$ $\exists SecureFileSystem$ $pid? : PROCID$ $o? : OBJECT$ $rep_0! : SFSREPORT$
$pid? \in \text{dom } aprocs$ $ocont \ o? \neq \langle \rangle$ $\neg osc \ o? \succeq (aprocs \text{ } pid?).supr$ $rep_0! = permissionDenied$

$WriteE3 \triangleq PidNotExist$

$WriteE \triangleq WriteE1 \vee WriteE2 \vee WriteE3$

$WriteOk \triangleq WriteOk1 \vee WriteOk2$

$Write \triangleq WriteOk \vee WriteE$

**end of Z Section** *write*

#### 4.17.1 Design and Implementation Comments

To optimize the security controls performed in *Write*, we must analyze what happens if the access class of *o?* is chagend between to *Writes*. The first conclusion is that it is not possible to perform those controls only at the first *Write*. A compromise occurs at least in the following case:

- Say processes *pf* and *pl* have open file *f* (with access class *c<sub>f</sub>*) for writing
- Assume that  $supr_{pf} \succ c_f \succ supr_{pl}$
- Let us suppose that *pf* writes into *f* before *pl*, then the system checks whether *pf* can do that, it decides that *pf* can, and records this fact for future times
- Now, if *pl* truncates *f* to zero<sup>5</sup> and writes something into it, *f*'s access class is downgraded to  $supr_{pl}$
- Hence, if *pf* writes once again into *f*, information of a higher class ( $supr_{pf}$ ) will be stored in a file with a lower class ( $supr_{pl}$ ) because the system will not check *f*'s access class again.

<sup>5</sup>To truncate a file can be done without compromise by any process on every file.

Hence, to optimize this checks it is necessary to record whether the process can write into a file and whether the access class of the file has changed since the last write. This can be implemented with one bit per process and file, say *may*. If *may* equals 1, write is allowed; otherwise security controls must be re-checked. Initially (when the file is open) *may* is set to 0, and right after the first successful write is set to 1 for this file and process. If the access class of an open file is changed, *may* is set to 0 for all the process that has the same file open in *write* mode. Once security controls are re-checked for a given file, *may* is set to 1. That this optimization is correct would deserve a formal proof.

This operation could be implemented jointly with *WriteLT*.

## Writing into Empty Objects

The semantics we have given to *Write* allows this kind of attacks:

1. User *h* with access class  $c_h$  ( $\succ L$ ) creates a file named *secret* with the intention of writing a secret document.
2. To write the document, *h* uses a text editor, say *te*, which is a Trojan horse (obviously *h* ignores this fact)
3. *h* has logged in recently, so his shell is working at *L* (see section 3.3)
4. *h* types in: *te secret* and press enter
5. Then *h* invokes *Chinsec* to upgrade his input to  $c_h$  (remember that he wants to write a secret file)
6. After a while he closes the session (so at this moment *secret*'s access class is  $c_h$ )
7. The next day, *h* continues to working on *secret* thus he types in from his shell: *te secret* and press enter (note that his input is at *L*)
8. Now, let us say he forget to upgrade his input with *Chinsec*, and *te* (which is a Trojan horse) truncates *secret* to zero and writes into it the new input typed in by *h*<sup>6</sup>
9. This new input is intended to be at  $c_h$  but in fact is at *L*, and so *secret* is declassified to *L*
10. Then the spy who installed *te* will be able to read from *secret* and write its content on his terminal

Clearly, this attack needs the sloppiness of users. We believe that a perfectly secure system cannot help much if it is used by sloppy users. In fact, with or without a computer system a sloppy user may disclose information any way. However, some countermeasures could be taken. For example, if part of the screen is reserved by the kernel to communicate with the user then, a warning message may be displayed showing the current access class of the input and/or every time a file is truncated.

## 4.18 WriteLT

**Description** Writes into a logical terminal

**Input parameters** *pid?* : *PROCID*

**Preconditions** *pid?* must be a valid process

---

<sup>6</sup>*h* may note this because the text written yesterday is not printed on his screen. However, *te* can mimic this too.

**Postconditions**  $pid?$ 's memory is written into the output stream of the logical terminal which  $pid?$  is connected to

This operation represents the **write** system call when the object to be written is a logical terminal connected to the process issuing the call. It may be implemented as part of the code of **write**. We have modeled it as a special case of *Write* because in our model logical terminals have a different type than objects.

We strongly recommend to read *Write* description before implementing this operation. Similar design considerations apply to this operation.

In what follows, note that **root** has no special privileges when requesting this operation. That is to say, **root** cannot violate *WriteLT*. More generally, **root** has no special privileges with respect to the MLS model.

## Z Section *writel*, parents: *state, definitions*

*WriteLT* allows a process to write from its memory into a logical terminal if the process is working at an access class dominated by the maximum access class of the physical terminal connected to the logical terminal. This precondition is encoded in the first **let** construct.

$WriteLTok$ $\Delta LogicalTerminals$ $\Xi UsersAndTerminals; \Xi Users; \Xi ProcessList; \Xi FileSystemObjects$ $\Delta LTCont$ $pid? : PROCID$ $repo! : SFSREPORT$
$pid? \in \text{dom } aprocs$ <b>let</b> $pt == (upt \ (aprocs \ pid?).usr) \bullet$ $\quad mptsc \ pt \succeq (aprocs \ pid?).supr$ <b>let</b> $lt == (aprocs \ pid?).lt \bullet$ $\quad stdout' = stdout \wedge writelt \ (aprocs \ pid?).mem$ $\quad \wedge stdin' = stdin$ $\quad \wedge ltcont' = ltcont \oplus \{lt \mapsto \theta LTCont'\}$ $ltsc' = ltsc$ $ttys' = ttys$ $repo! = ok$

It is worth noticing that the access class of the logical terminal is not considered in order to authorize the operation (while it may be lower than  $mptsc$ ). This is so because the kernel interface does not offer an operation to retrieve what was written into a logical terminal. Thus, we have to take care only about who can *see* what is written, and not if a process can *read* that. In other words, the current access class of the physical terminal (which is equal to the access class of the logical terminal) applies only to the input entered by the user, and not to the output seen by him.

*WriteLT*'s postcondition is set in the second **let** : some portion of the process' memory is added to the output buffer of the logical terminal.

There are two possible error conditions:  $pid?$  is not a valid process or the maximum access class of  $pt$  does not dominate the working access class of  $pid?$ .



$WriteLTE1$ $\exists SecureFileSystem$ $pid? : PROCID$ $rep0! : SFSREPORT$
$pid? \in \text{dom } aprocs$ $\text{let } pt == (upt \ (aprocs \ pid?).usr) \bullet \neg \text{mptsc } pt \succeq (aprocs \ pid?).supr$ $rep0! = permissionDenied$

$WriteLTE2 \triangleq PidNotExist$

$WriteLTE \triangleq WriteLTE1 \vee WriteLTE2$

$WriteLT \triangleq WriteLTOk \vee WriteLTE$

**end of Z Section** *writelte*

## 4.19 The Interface to be Used by Processes

This section contains a schema defining the interface that processes must use.

**Z Section** *pcop, parents:* *chobjsc, chsubsc, close, create, exec, fork, link, links, mmap, open, oscstat, read, t*

$ProcessControlledOperations \triangleq$

$Chobjsc$   
 $\vee Chsubsc$   
 $\vee Close$   
 $\vee Create$   
 $\vee Exec$   
 $\vee Fork$   
 $\vee Link$   
 $\vee LinkS$   
 $\vee Mmap$   
 $\vee Open$   
 $\vee Oscstat$   
 $\vee Read$   
 $\vee ReadLT$   
 $\vee Rename$   
 $\vee Setuid[newuid?/new?]$   
 $\vee Stat$   
 $\vee Write$   
 $\vee WriteLT$

**end of Z Section** *pcop*

## 4.20 Other operations

The following system calls must be implemented as they currently are in Lisex:

- `acladd`

- `acldel`
- `chmod`
- `chown`
- `munmap`

System call `owner_close` must not be implemented at all; system call `sscstat` must be implemented as a library function (because users' access classes are stored in a file inside the TCB).

## Chapter 5

# Operations Controlled by the System

In this chapter we describe all the operations controlled by the system. We strongly recommend to read the introduction to chapter 3. System controlled operations are implemented as kernel internal actions.

### 5.1 Get

**Description** Input from the physical terminal is copied to the logical terminal attached to it

**Input parameters**  $pt? : PTERM$

**Preconditions** New input must be available in  $pt?$

**Postconditions** Input available in  $pt?$  is added to the end of the input buffer of the logical terminal attached to  $pt?$

With this operation we tried to describe the transfer of characters between a physical terminal and the logical terminal attached to it. This transfer is made on a character by character basis. Every time a user pushes a key, variable *ready* is set to a non-negative value, and every time the kernel process this character it sets *ready* to a negative value. This description is not intended to be a formalization of the actual Linux behavior, rather our intention was to describe some (very) abstract properties that must be enforced.

**Z Section** *get*, **parents:** *state*, *definitions*

We need a couple of schemas to promote operations of *PhysicalTerminal* and *LTCont*. *PTGet* simply says that the system has processed the input available in a physical terminal. *LTAdd* adds a *CCHAR* to the input buffer of a logical terminal.

*PTGet*

$\Delta PhysicalTerminal$

$ready' = -1$

$input' = input$

$output' = output$

<i>LTCAdd</i>
$\Delta LTCont$
$cc : CCHAR$
$stdin' = stdin \frown \langle cc \rangle$
$stdout' = stdout$

The system should pay attention only to physical terminals that are being used ( $pt? \in \text{ran } upt$ ), and only when there is new input available ( $0 \leq (pts \ pt?).ready$ ) on them. Note that the input parameter  $pt?$  is not provided by the environment because this operation is intended to be initiated by the system.

<i>GetOk</i>
$\Delta UsersAndTerminals$
$\Delta LogicalTerminals$
$\Xi Users; \Xi FileSystemObjects; \Xi ProcessList$
$PTGet$
$LTCAdd$
$pt? : PTERM$
$rep_0! : SFSREPORT$
$pt? \in \text{ran } upt$
$0 \leq (pts \ pt?).ready$
$pts' = pts \oplus \{pt? \mapsto \theta PhysicalTerminal'\}$
$cc = (pts \ pt?).input$
$ltcont' = ltcont \oplus \{ttyspt? \mapsto \theta LTCont'\}$
$upt' = upt$
$cptsc' = cptsc$
$ltsc' = ltsc$
$ttys' = ttys$
$rep_0! = ok$

If everything is right the new character is added to the end of the input buffer of the logical terminal connected to  $pt?$ .

Errors may arise if  $pt?$  is not being used or if there is no new input to process.

<i>GetE1</i>
$\Xi SecureFileSystem$
$pt? : PTERM$
$rep_0! : SFSREPORT$
$(pts \ pt?).ready < 0$
$rep_0! = noInput$

$GetE2 \triangleq TerminalNotUsed$

$GetE \triangleq GetE1 \vee GetE2$

$Get \triangleq GetOk \vee GetE$

end of Z Section *get*

## 5.2 Put

**Description** The system takes data from the output buffer of a logical terminal and writes this data on the corresponding physical terminal

**Input parameters** *lt?* : *LTERM*

**Preconditions** *lt?* must be in use and there must be something to write on the physical terminal

**Postconditions** The first character in the output buffer of *lt?* is written on the output device of *ttys lt?*

With this operation we tried to describe the transfer of characters from a logical terminal to the physical terminal attached to it. This transfer is made on a character by character basis. It may be that this modelization is not an accurate description of the real process, but certainly it captures its essence. In fact, this operation is not critical and is here just for completeness. The only thing that matters is that the system must send data taken from a logical terminal only to the physical terminal attached to it. The exact way in which this is accomplished is unimportant.

**Z Section *put*, parents:** *state, definitions*

We start with a schema at the *PhysicalTerminal* level describing how its output device is updated.

<i>PTPut</i>
$\Delta PhysicalTerminal$
<i>cc</i> : <i>CCHAR</i>
$output' = cc$
$ready' = ready$
$input' = input$

The first character available in the output buffer of a logical terminal is removed.

<i>LTCRemove</i>
$\Delta LTCont$
$stdout' = tail\ stdout$
$stdin' = stdin'$

Output is sent to the physical terminal attached to *lt?* whenever there is output available and *lt?* is being used. See how output is sent only to the physical terminal attached to *lt?*: *pt == (ttys lt?)*.

---

*PutOk*

$\Delta UsersAndTerminals$

$\Delta LogicalTerminals$

$\Xi Users; \Xi FileSystemObjects; \Xi ProcessList$

*PTPut*

*LTCRemove*

$lt? : LTERM$

$repo! : SFSREPORT$

---

**let**  $pt == (ttys \sim lt?) \bullet$   
 $pt \in \text{ran } upt$   
 $\wedge (pts \ pt) = \theta PhysicalTerminal$   
 $\wedge (ltcont \ lt?).stdout \neq \langle \rangle$   
 $\wedge cc = head(ltcont \ lt?).stdout$   
 $\wedge pts' = pts \oplus \{pt \mapsto \theta PhysicalTerminal'\}$   
 $ltcont' = ltcont \oplus \{lt? \mapsto \theta LTCont'\}$   
 $upt' = upt$   
 $cptsc' = cptsc$   
 $ltsc' = ltsc$   
 $ttys' = ttys$   
 $repo! = ok$

---

There are two possible errors:  $lt?$  is not being used or there is no output available in it.

---

*PutE1*

$\Xi SecureFileSystem$

$lt? : LTERM$

$repo! : SFSREPORT$

---

$(ltcont \ lt?).stdout = \langle \rangle$   
 $repo! = noOutput$

---

$PutE2 \triangleq [TerminalNotUsed; lt? : LTERM \mid pt? = ttys \sim lt?]$

$PutE \triangleq PutE1 \vee PutE2$

$Put \triangleq PutOk \vee PutE$

**end of Z Section** *put*

## 5.3 System Internal Operations

This section contains a schema defining all the systema internal operations.

**Z Section** *scop*, **parents:** *get*, *put*

$SystemControlledOperations \triangleq Get \vee Put$

**end of Z Section** *scop*

## Chapter 6

# The Transition Relation

This chapter contains just a schema consisting of the disjunction of all the possible operations. We write it through the three interfaces defined in the previous chapters.

**Z Section** *tranrel*, **parents:** *ucop*, *pcop*, *scop*

$$\begin{aligned} \textit{TransitionRelation} &\hat{=} \\ &\textit{UserControlledOperations} \\ &\vee \textit{ProcessControlledOperations} \\ &\vee \textit{SystemControlledOperations} \end{aligned}$$

**end of Z Section** *tranrel*

## Chapter 7

# Formal Security Model

In this section we state properties the system must enforce. Properties are state predicates and the intention is to prove that they are state invariants. There are properties of particular state schemas and properties that relate two or more schemas.

### 7.1 Invariants of *UsersAndTerminals*

The first property regarding users and terminals says that if a physical terminal is working at a particular current access class then, every input on that terminal must be taken at the same access class.

**Z Section** *secmod*, **parents:** *tranrel*

<i>UTInvI</i>	
<i>UsersAndTerminals</i>	
$\forall pt : PTERM \bullet cpts\ pt = (pts\ pt).input.2$	

There is a similar property for output written on the terminal. It says that characters written on a terminal must have a classification dominated by the maximum access class of that terminal.

<i>UTInvO</i>	
<i>UsersAndTerminals</i>	
$\forall pt : PTERM \bullet mpts\ pt \succeq (pts\ pt).output.2$	

The last property regarding physical terminals says that a terminal could not work at an access class not dominated by its maximum access class.

<i>UTInvCSC</i>	
<i>UsersAndTerminals</i>	
$\forall pt : PTERM \bullet mpts\ pt \succeq cpts\ pt$	

Finally, all invariants are gathered in one schema.

$$UTInv \triangleq UTInvI \wedge UTInvO \wedge UTInvCSC$$



## 7.2 Invariants of *LogicalTerminals*

There is only one invariant that we can express considering the state of *LogicalTerminals*; other important properties relate physical and logical terminals, see section 7.6. The invariant says that the access class of the input stored in a logical terminal must equal the access class of the terminal. It would be possible to state a weak version by asking  $\succeq$  rather  $=$ , but that would be not implementable. Precisely, at implementation level the system does not manage *CCHARs* but *CHARs* thus, if a logical terminal stores input (*CHAR*) at different access classes then it would be impossible to manage it securely. In other words, it would be impossible for the system to write input in a file preserving a secure information flow.

$LTInv$
<i>LogicalTerminals</i>
$\forall lt : LTERM \bullet$ $\quad \forall cc : CCHAR \mid cc \in \text{ran}(ltcont \ lt).stdin \bullet cc.2 = ltsc \ lt$

## 7.3 Invariants of *Users*

The invariant of this schema is just a *well-formedness* property. The set of users recognized by the system must equals the set of users who have access classes, and only registered users may belong to groups of users. The reason to ask for  $=$  and not  $\subseteq$  is that there is a group, called *allgrp*, that contains all the users of the system.

$UInvWF$
<i>Users</i>
$users = \text{dom } usc$ $\bigcup \{g : \text{ran } grps\} = users$

$$UInv \triangleq UInvWF \wedge USCInv$$

## 7.4 Invariants of *FileSystemObjects*

Again, the first invariant of this type is a well-formedness property. The second line says that every object in the system is owned by *rootgrp*.

$FSOInvWF$
<i>FileSystemObjects</i>
$objs = \text{dom } osc = \text{dom } oacl = \text{dom } ocont$ $\forall o : OBJECT \mid o \in objs \bullet (grp \ rootgrp) \mapsto OWNER \in (oacl \ o).acl$ $softtcb \subseteq objs$

*FSOInvIF* is one of the most important properties to be enforced. It says that an object must contain data as sensitive as the object's classification. Here the reader may notice the importance of considering *CCHARs* and not plain *CHARs* in order to be able to prove important properties. However, given that at implementation level there are only *CHARs*, the system must implement a

simplified version of this property: all characters stored in an object  $o$  will have  $o$ 's access class, but they can be stored in it only if they come from a source with an access class dominated by  $o$ 's.

$FSOInvIF$
$FileSystemObjects$
$\forall o : OBJECT \mid o \in objs \bullet$ $osc\ o \succeq SUP\{cc : CCHAR \mid cc \in ran(ocont\ o) \bullet cc.2\}$

$$FSOInv \cong FSOInvWF \wedge FSOInvIF$$

## 7.5 Invariants of Process

There is only one property for this type but it is a crucial one. It states that the kernel must control every process  $p$  in order to record in  $p.supr$  the least upper bound of the access classes of the information it has been reading.

Given that processes are outside of our TCB, and that they have complete control of their memory spaces, then the system must keep track of how high is the information that each process has read. In this way, afterwards, the system will be able to deny certain write requests of certain processes. By a write request we understand calling the *Write* system call.

$PInv$
$Process$
$supr = SUP\{cc : CCHAR \mid cc \in ran\ mem \bullet cc.2\}$

## 7.6 Secure File System Properties

In this section we account for those invariants that relate two or more components of the environment. The first one is another well-formedness property relating the state of each process with the other components of the environment.

$SFSInvWF$
$Users$
$ProcessList$
$FileSystemObjects$
$\forall p : Process \mid p \in ran\ aprocs \bullet$ $p.usr \in users$ $\wedge p.suid \in users$ $\wedge p.or \subseteq objs$ $\wedge p.ow \subseteq objs$ $\wedge p.mmfr \subseteq p.or$ $\wedge p.mmfw \subseteq p.ow$ $\wedge p.prog \in objs$

The property formalized below is essential to the security of the system. Users can work on terminals with access class dominated by their own. In other words, low level users are not permitted

to enter where high level users work. If output devices (such as printers) are considered as particular kinds of physical terminals, *SFSInvUT* says that a user cannot enter a printer's room if higher level users print on it.

<i>SFSInvUT</i>
<i>UsersAndTerminals</i>
<i>Users</i>
$\forall u : USER \mid u \in \text{dom } \text{upt} \bullet \text{usc } u \succeq \text{mptsc } (\text{upt } u)$

Property *SFSInvUT* in conjunction with property *PTInvO* ensures that a user cannot inadvertently disclose information.

Next, we state two relations between physical and logical terminals. A the first one says that a logical terminal must operate at the current access class of the physical terminal attached to it.

<i>SFSInvPLTI</i>
<i>UsersAndTerminals</i>
<i>LogicalTerminals</i>
$\forall pt : PTERM \bullet \text{cptsc } pt = \text{ltsc } (\text{ttys } pt)$

However, the output sent by a logical terminal to its physical terminal may be higher than *cptsc*, in fact it must be lower than *mptsc*. Given that everything stored in the output buffer of a logical terminal is sent eventually to its physical terminal, we require that the least upper bound between all *CCHAR* stored in *output* be dominated by *mptsc*.

<i>SFSInvPLTO</i>
<i>UsersAndTerminals</i>
<i>LogicalTerminals</i>
$\forall pt : PTERM \bullet$ $\text{mptsc } pt \succeq \text{SUP}\{cc : CCHAR \mid \langle cc \rangle \text{ in } (\text{ltcont } (\text{ttys } pt)).\text{stdout} \bullet cc.2\}$

We close this section with a schema composed by the conjunction of all the properties introduced up to here.

$$\begin{aligned}
SFSInv &\triangleq \\
&UTInv \\
&\wedge LTIInv \\
&\wedge UIInv \\
&\wedge FSOInv \\
&\wedge PInv \\
&\wedge SFSInvWF \\
&\wedge SFSInvUT \\
&\wedge SFSInvPLTI \\
&\wedge SFSInvPLTO
\end{aligned}$$

## 7.7 The Missed Property

The fundamental property of the system should be that any input done at a particular classification must not be outputted at a lower classification. In other words, if 'a' is entered through a terminal

working at current access class  $c_i$ , then it cannot be written (in any future time) on a terminal with maximum access class  $c_o$ , with  $c_o \not\geq c_i$ . But this kind of property is not easy to express in a language such as Z. It would be necessary a language with enough expressive power as a modal or temporal logic.

However, validity of properties *UTInvO*, *PInv* and *FSOInvIF*, given what we have included inside the TCB, and Tables 2.1 to 2.5, should be enough to convince anyone that the fundamental property stated above could be proved if we were able to write it down.

The argument is as follows:

1. Physical terminals properly classify the input they receive (TCB)
2. The file system program does not change the classification of any input (or at least it does not lower the access class of inputs) (TCB and *FSOInvIF*)
3. The operating system kernel properly classify the memory of each process (*PInv*)
4. The operating system kernel controls the information flow across objects and memory (*SFSInvIF* and TCB)
5. The operating system kernel controls what information is written on physical terminals (*UTInvO*)

However, we can write a predicate that state a simplified version of the intended property rather precisely (we borrowed this technique from [8]):

$$\text{TranquilityPrinciple} \triangleq \text{TransitionRelation} \wedge \neg (\text{Chobjsc} \vee \text{Chsubsc} \vee \text{Input})$$

**theorem** NoIllegalFlow

$$\begin{aligned} & \forall tr : \text{seq } \text{SecureFileSystem} \bullet \\ & \quad tr(1) \in \{ \text{SFSInit} \bullet \theta \text{SecureFileSystem} \} \\ & \quad \wedge tr(1) \mapsto tr(2) \in \{ \text{Input} \bullet \theta \text{SecureFileSystem} \mapsto \theta \text{SecureFileSystem}' \} \\ & \quad \wedge (\forall i : 2 \dots \#tr - 1 \bullet \\ & \quad \quad tr(i) \mapsto tr(i+1) \\ & \quad \quad \in \{ \text{TranquilityPrinciple} \bullet \theta \text{SecureFileSystem} \mapsto \theta \text{SecureFileSystem}' \}) \\ & \Rightarrow (\exists pt : \text{PTERM} \bullet \\ & \quad (pt, tr(1)) \mapsto tr(2) \in \{ \text{Input} \bullet (pt?, \theta \text{SecureFileSystem}) \mapsto \theta \text{SecureFileSystem}' \} \\ & \quad \Rightarrow (\forall n : 3 \dots \#tr; ptout : \text{PTERM}; sc : \text{SecClass} \bullet \\ & \quad \quad \text{let } ptin == ((tr\ 2).pts\ pt).input \bullet \\ & \quad \quad \neg ptin.2 \succeq sc \\ & \quad \quad \Rightarrow (ptin.1, sc) \neq ((tr\ n).pts\ ptout).output)) \end{aligned}$$

**end of Z Section** *secmod*

## 7.8 Simple Security

Simple security is a property formalized in [2, 3]. It is stated as follows:

(BLP) If subject  $s$  with access class  $c_s$  has opened object  $o$  with access class  $c_o$  then,  $c_s \succeq c_o$ .

The intention behind this property is to fulfil the fundamental requirement of the DoD's security policy<sup>1</sup>:

---

<sup>1</sup>DoD is Department of Defense (of the United States of America).

(DoD) Person  $p$  with clearance  $c_p$  may read document  $d$  with classification<sup>2</sup>  $c_d$  if and only if  $c_p$  dominates  $c_d$  [9].

In requirement engineering, (DoD) is a requirement and (BLP) is its specification [13]. We want to implement (DoD) in a different way because we consider that (BLP) is unnecessary restrictive. As stated, (DoD) says nothing about processes, files, computer memory, and so on. It only talks about persons, documents and certain access attributes of them. If we succeed in implementing a system that prevents *persons* to *see* information they are unauthorized to see, then our system obeys (DoD).

In our model, persons are elements of *USER* (they are not processes), and users can see information only on their physical terminals. If we build a system that never writes information on a physical screen when an a user not unauthorized to see it is seated in front of this terminal, then we have a secure system<sup>3</sup>. Moreover, nobody should matter about what the system does with characters, files and processes: it could merge files in strange ways, it could manage processes in bizarre ways.

We are strongly convinced that if our system verify *SFSInv* then the previous situation will be impossible.

On the other hand, by not implementing (BLP) we are allowing that higher files be contaminated with lower data. But this is an integrity problem, it does not compromise confidentiality. Integrity will not be assured by implementing (BLP) [4]. Moreover, some tasks and features of Linux will be easier to implement with an appropriate configuration. Consider, for example, `/dev/null` or how to make backups. We believe that in doing so trusted processes will be seldom needed.

## 7.9 Where Can Users Work?

In our model users can log in on terminals not trusted as they. One may be tempted to impose stronger restrictions on where users can work. For example, we could have stated that users can work only at terminals with their access classes. The reason to impose such a restriction is based on the fact that, otherwise, we left a door open to some attacks regarding the authentication of users to the system. A possible scenario is as follows.

- Let us say user  $u$  with access class  $c_u$  is willing to log in on physical terminal  $pt$  with maximum access class  $c_{pt}$ , where  $c_u \succeq c_{pt}$ .
- $pt$  has this access class because it is exposed to certain attacks. For example,  $pt$  lays in a public place, or it is close to a window or outside a TEMPEST room; moreover,  $pt$ 's hardware could had been built by a company not trusted enough.
- The secret used by  $u$  to authenticate to the system must be as trusted as himself, so it must be classified at  $c_u$ . If this is not true, then, for example,  $u$  may be careless in protecting this secret.
- In order to authenticate to the system  $u$  has to show his secret to  $pt$ . Here, to show means to write a password, to use a piece of  $pt$ 's hardware to calculate a key, to enter a PIN, etc.
- Hence, if  $pt$  is not trusted as  $u$ , then  $u$ 's secret could be inadvertently disclosed by  $u$  or purposely stolen by  $pt$  or an attacker with access to  $pt$ 's room.
- Note that, once  $u$  has logged in, the system will not write on  $pt$  information with an access class not dominated by  $c_{pt}$  even if  $u$  request such an action.

However, by imposing stronger a restrictions as the one stated above we cannot avoid this scenario: a user can always go and try to log in on a non trusted terminal giving the chance to an attacker to

---

<sup>2</sup>Clearance and classification are synonymous of access class.

<sup>3</sup>Obviously, physical security must work too.

steal his or her authentication secret<sup>4</sup>. In consequence, imposing such a restriction will not make the system more secure but it certainly make it less usable.

---

<sup>4</sup>Tahnks to Felipe Manzano for noticing this fact.

## Chapter 8

# Subject Security Classes

This chapter describes the relationship between users and access classes. Every user has a unique security class. Moreover, new users may be added to the system and users may have their security classes changed by MAC administrators. Thus, we model this relation as a partial function from *USER* onto *SecClass*. This relation must be implemented as an abstract data type (ADT).

### 8.1 Basic Types, Parameters, and State Definition

**Z Section** *subjectsc*, **parents:** *main*, *sc*

*uscOk*  $\approx$  is returned when there are no errors in the invocation of some operation

*uscError*  $\approx$  is returned when a non previously specified error occurs in the invocation of some operation

$$USCREPORT ::= uscOk \mid uscError$$

*secadm*  $\approx$  is the MAC administrator delivered with the system

*SECADMIN*  $\approx$  is a category reserved for those users enabled to change security classes, i.e. MAC administrators

$$\left| \begin{array}{l} secadm : USER \\ SECADMIN : CATEGORY \end{array} \right.$$

As we said above, the relation between users and their security classes is modeled as a partial function.

$$\boxed{\begin{array}{l} \text{UserSecClass} \\ \hline usc : USER \leftrightarrow SecClass \end{array}}$$

The invariant for this ADT says that *secadm* cannot be removed, and that if a user has category *SECADMIN*, then this must be the only one category in her or his access class.

$USCInv$	
$UserSecClass$	
$secadm \in \text{dom } usc$	
$(usc \ secadm).categs = \{SECADMIN\}$	
$\forall u : USER$	
$u \in \text{dom } usc \bullet$	
$SECADMIN \in (usc \ u).categs \Rightarrow (usc \ u).categs = \{SECADMIN\}$	

Initially the ADT is in a state that, by definition, verifies the invariant.

$$USCInit \triangleq USCInv$$

## 8.2 Operations

We will describe the operations on *UserSecClass* in part by promoting operations of *SecClass* (see chapter 9). Thus, we start this section by introducing the appropriate framing schema for operation promotion [12, 10]. This schema defines how *usc* must be updated when a *SecClass* operation is invoked from this level. The last *D* in the schema name stands for *Delta*, that is, this framing schema is used just for operations that change the state. Latter, another framing schema will be defined for those operations that consult the state.

$SecClassToUserSecClassD$	
$\Delta SecClass$	
$\Delta UserSecClass$	
$u? : USER$	
$rep_1! : USCREPORT$	
$u? \in \text{dom } usc$	
$(usc \ u?) = \theta SecClass$	
$usc' = usc \oplus \{u? \mapsto \theta SecClass'\}$	
$rep_1! = uscOk$	

The schema above is intended to be used only in successful cases, hence we need to define schemas for the error cases. We have one implicit error case, when a *SecClass* operation fails, and one explicit when user *u?* does not exist.

$$USCErrorReport \triangleq [\exists UserSecClass; rep_1! : USCREPORT \mid rep_1! = uscError]$$

$$USCUserNotExist \triangleq [\exists UserSecClass; u? : USER \mid u? \notin \text{dom } usc]$$

The following operation sets the level of the access class of a given user. It is specified by promoting *SCSetLevel*. Note how in the third case we take into account all of the possible failures of *SCSetLevel*.

$$USCSetLevelOk \triangleq SecClassToUserSecClassD \wedge SCSetLevelOk$$

$$USCSetLevelE1 \triangleq SCSetLevel \wedge USCUserNotExist \wedge USCErrorReport$$

$$USCSetLevelE2 \triangleq SCSetLevelE \wedge USCErrorReport$$

$$USCSetLevelE \triangleq USCSetLevelE1 \vee USCSetLevelE2$$

$$USCSetLevel \triangleq USCSetLevelOk \vee USCSetLevelE$$



The addition of a category to the access class of a given user cannot be described just by promoting  $SCAddCat$  because at this level we must see whether  $SECADMIN$  category is to be added or not. Note that the category set of  $secadm$  cannot be changed; this precondition is redundant given the second one but we believe it is a good idea to reinforce this property.

$USCAddCatOk$
$SecClassToUserSecClassD$
$SCAddCatOk$
$u? \neq secadm$
$c? = SECADMIN \Rightarrow (usc\ u?).categs = \emptyset$

$$USCAddCatE1 \triangleq SCAddCat \wedge USCUserNotExist \wedge USSErrorReport$$

$USCAddCatE2$
$SCAddCat$
$USSErrorReport$
$u? : USER$
$c? : CATEGORY$
$u? = secadm \vee (c? = SECADMIN \wedge (usc\ u?).categs \neq \emptyset)$

$$USCAddCatE3 \triangleq SCAddCatE \wedge USSErrorReport$$

$$USCAddCatE \triangleq USCAddCatE1 \vee USCAddCatE2 \vee USCAddCatE3$$

$$USCAddCat \triangleq USCAddCatOk \vee USCAddCatE$$

Now, we introduce an operation that sets the level and the category set at the same time. Again, a little bit of extra preconditions should be considered.

$USCSetSCOk$
$SecClassToUserSecClassD$
$SCSetSCOk$
$u? \neq secadm$
$SECADMIN \in C? \Rightarrow ((usc\ u?).categs = \emptyset \wedge C? = \{SECADMIN\})$

$$USCSetSCE1 \triangleq SCSetSC \wedge USCUserNotExist \wedge USSErrorReport$$

$USCSetSCE2$
$SCSetSC$
$USSErrorReport$
$u? : USER$
$C? : \mathbb{P}\ CATEGORY$
$l? : \mathbb{Z}$
$u? = secadm$
$\vee (SECADMIN \in C?$
$\wedge ((usc\ u?).categs \neq \emptyset \vee C? \neq \{SECADMIN\}))$

$$USCSetSCE3 \triangleq SCSetSCE \wedge USCErrorReport$$

$$USCSetSCE \triangleq USCSetSCE1 \vee USCSetSCE2 \vee USCSetSCE3$$

$$USCSetSC \triangleq USCSetSCOk \vee USCSetSCE$$

Below we define the framing schema for promoting operations that consult the state; the  $X$  at the end of the name stands for  $Xi$  (i.e.  $\Xi$ ).

$SecClassToUserSecClassX$	_____
$\Xi SecClass$	
$\Xi UserSecClass$	
$u? : USER$	
$rep_1! : USCREPORT$	
$u? \in \text{dom } usc$	
$(usc \ u?) = \theta SecClass$	
$usc' = usc$	
$rep_1! = uscOk$	

The rest of this section describes the promotion of operations that consult the state; their names are self explanatory. The last schema defines the ADT's interface.

$$USCGetSizeOk \triangleq SecClassToUserSecClassX \wedge SCGetSize$$

$$USCGetSizeE \triangleq USCUserNotExist \wedge USCErrorReport$$

$$USCGetSize \triangleq USCGetSizeOk \vee USCGetSizeE$$

$$USCGetCatOk \triangleq SecClassToUserSecClassX \wedge SCGetCat$$

$$USCGetCatE \triangleq USCUserNotExist \wedge USCErrorReport$$

$$USCGetCat \triangleq USCGetCatOk \vee USCGetCatE$$

$$USCGetLevelOk \triangleq SecClassToUserSecClassX \wedge SCGetLevel$$

$$USCGetLevelE \triangleq USCUserNotExist \wedge USCErrorReport$$

$$USCGetLevel \triangleq USCGetLevelOk \vee USCGetLevelE$$

$$USCInterface \triangleq$$

$$USCGetLevel$$

$$\vee USCGetSize$$

$$\vee USCGetCat$$

$$\vee USCSetLevel$$

$$\vee USCAddCat$$

$$\vee USCSetSC$$

### 8.3 Proof Obligations

**theorem** USCSetLevelPI

$$USCInv \wedge USCSetLevel \Rightarrow USCInv'$$

**theorem** USCAddCatPI

$$USCInv \wedge USCAddCat \Rightarrow USCInv'$$

**theorem** USCSetSCPI

$$USCInv \wedge USCSetSC \Rightarrow USCInv'$$

**theorem** USCInterfacePI

$$USCInv \wedge USCInterface \Rightarrow USCInv'$$

**end of Z Section** *subjectsc*

## Chapter 9

# Security Classes

In this chapter we describe security or access classes (SC). Usually a SC is represented as an ordered pair which first component is called *level* and the second is a set of *categories* (see [9, 5] for more details). SCs should be implemented as an ADT where the hidden data structure will be an implementation of the state schema, and the interface will comprise the state operations defined below.

### 9.1 Basic Types, Parameters, and State Definition

**Z Section** *sc*, parents: *toolkit*

*CATEGORY*  $\approx$  all the possible categories, departments or need-to-know

*scCatFull*  $\approx$  is returned when the size of the set of categories reaches its maximum capacity

*scOk*  $\approx$  is returned when there are no errors in the invocation of some operation

*scError*  $\approx$  is returned when a non previously specified error occurs in the invocation of some operation

[*CATEGORY*]

*SCREPORT* ::= *scCatFull* | *scOk* | *scError*

*MAXLEVEL*  $\approx$  maximum possible value of a security level

*MAXNCAT*  $\approx$  maximum size of a category set

$MAXLEVEL, MAXNCAT : \mathbb{N}$
$MAXNCAT > 0$

We model a SC as a schema comprising to variables with obvious meanings.

<i>SecClass</i>
<i>level</i> : $\mathbb{Z}$
<i>categs</i> : $\mathbb{F} \text{ } CATEGORY$

The ADT's invariant says that the *level* of any *SecurityClass* must belong to a finite interval, and that the size of the set of categories must be less or equal to *MAXNCAT*.

$SCInv$
$SecClass$
$level \in 0 \dots MAXLEVEL$
$\#categs \leq MAXNCAT$

### Domain check proof

prove by reduce;  
end proof.

Now, we define the standard partial order over the set of access classes. The symbol  $\succeq$  it is read *dominates*.

**syntax**  $\succeq$  *inrel*

$\_ \succeq \_ : SecClass \leftrightarrow SecClass$
$\forall x, y : SecClass \bullet$ $x \succeq y \Leftrightarrow x.level \geq y.level \wedge y.categs \subseteq x.categs$

$SUP$  is the least upper bound operator on the set of security classes [7]. We define it with domain on  $\mathbb{P} SecClass$  and  $Sup$  with domain on  $SecClass \times SecClass$ . Similarly, the greatest lower bound operators are defined ( $INF$  and  $Inf$ ).

$SUP : \mathbb{P}_1 SecClass \rightarrow SecClass$
$\forall SC : \mathbb{P}_1 SecClass \bullet$ $(SUP SC).level = \max\{s : SecClass \mid s \in SC \bullet s.level\}$ $\wedge (SUP SC).categs = \bigcup\{s : SecClass \mid s \in SC \bullet s.categs\}$
$Sup : SecClass \rightarrow SecClass \rightarrow SecClass$
$\forall sc_1, sc_2 : SecClass \bullet$ $(Sup sc_1 sc_2).level = \text{if } sc_1.level \geq sc_2.level \text{ then } sc_1.level \text{ else } sc_2.level$ $\wedge (Sup sc_1 sc_2).categs = sc_1.categs \cup sc_2.categs$
$INF : \mathbb{P} SecClass \rightarrow SecClass$
$\forall SC : \mathbb{P} SecClass \bullet$ $(INF SC).level = \min\{s : SecClass \mid s \in SC \bullet s.level\}$ $\wedge (INF SC).categs = \bigcap\{s : SecClass \mid s \in SC \bullet s.categs\}$
$Inf : SecClass \rightarrow SecClass \rightarrow SecClass$
$\forall sc_1, sc_2 : SecClass \bullet$ $(Inf sc_1 sc_2).level = \text{if } sc_1.level \geq sc_2.level \text{ then } sc_2.level \text{ else } sc_1.level$ $\wedge (Inf sc_1 sc_2).categs = sc_1.categs \cap sc_2.categs$

On the initial state a security class equals  $L$ , i.e. the lower bound of the set of access classes.

$SCInit$
$SecClass$
$level = 0$
$categs = \emptyset$

$$L \triangleq SCInit$$

## 9.2 Operations

*SCGetSize* returns the number of categories in a given access class.

<i>SCGetSize</i>	_____
$\Xi SecClass$	
$size! : \mathbb{N}$	
$rep! : SCREPORT$	
$size! = \#categs$	
$rep! = scOk$	

*SCGetCat* returns a list with the catagories of a given access class.

<i>SCGetCat</i>	_____
$\Xi SecClass$	
$lcategs! : seq\ CATEGORY$	
$rep! : SCREPORT$	
$ran\ lcategs! = categs$	
$\#lcategs! = \#categs$	
$rep! = scOk$	

*SCGetLevel* returns the level of a given access class.

<i>SCGetLevel</i>	_____
$\Xi SecClass$	
$l! : \mathbb{Z}$	
$rep! : SCREPORT$	
$l! = level$	
$rep! = scOk$	

*SCSetAddCat* adds a category to the category set of an access class whenever the current amount of categories do not equals *MAXNCAT*. The other precondition ( $c? \notin categs$ ) is there just to warn the programer who will not have a set at implementation level.

<i>SCAddCatOk</i>	_____
$\Delta SecClass$	
$c? : CATEGORY$	
$rep! : SCREPORT$	
$c? \notin categs$	
$\#categs < MAXNCAT$	
$categs' = categs \cup \{c?\}$	
$level' = level$	
$rep! = scOk$	

There are two possible errors: when *categs* is full and when an existing category is to be added.

$SCAddCatE1$ $\exists SecClass$ $c? : CATEGORY$ $rep! : SCREPORT$
$c? \in categs$ $rep! = scError$

$SCAddCatE2$ $\exists SecClass$ $rep! : SCREPORT$
$\#categs = MAXNCAT$ $rep! = scCatFull$

The total operation is summarized below.

$$SCAddCatE \triangleq SCAddCatE1 \vee SCAddCatE2$$

$$SCAddCat \triangleq SCAddCatOk \vee SCAddCatE$$

$SCSetLevel$  sets the level of an access class whenever the input level lays in the appropriate interval.

$SCSetLevelOk$ $\Delta SecClass$ $l? : \mathbb{Z}$ $rep! : SCREPORT$
$0 \leq l? \leq MAXLEVEL$ $level' = l?$ $categs' = categs$ $rep! = scOk$

$SCSetLevelE$ $\exists SecClass$ $l? : \mathbb{Z}$ $rep! : SCREPORT$
$l? < 0 \vee MAXLEVEL < l?$ $rep! = scError$

$$SCSetLevel \triangleq SCSetLevelOk \vee SCSetLevelE$$

The following operation allows to set both the level and the set of categories at the same time. Its preconditions are obvious if  $SCAddCat$  and  $SCSetLevel$  have been read.

$SCSetSCOk$ $\Delta SecClass$ $l? : \mathbb{Z}$ $C? : \mathbb{F} \text{ CATEGORY}$ $rep! : SCREPORT$
$0 \leq l? \leq MAXLEVEL$ $\#C? \leq MAXNCAT$ $level' = l?$ $categs' = C?$ $rep! = scOk$

$SCSetSCE1 \triangleq SCSetLevelE$

$SCSetSCE2$ $\Xi SecClass$ $C? : \mathbb{F} \text{ CATEGORY}$ $rep! : SCREPORT$
$\#C? > MAXNCAT$ $rep! = scError$

$SCSetSCE \triangleq SCSetSCE1 \vee SCSetSCE2$

$SCSetSC \triangleq SCSetSCOk \vee SCSetSCE$

The interface of this ADT is summarized below.

$SCInterface \triangleq$   
 $SCGetLevel$   
 $\vee SCGetSize$   
 $\vee SCGetCat$   
 $\vee SCSetLevel$   
 $\vee SCAddCat$   
 $\vee SCSetSC$

### 9.3 Proof Obligations

**theorem**  $SCSetLevelPI$   
 $SCInv \wedge SCSetLevel \Rightarrow SCInv'$

**theorem**  $SCAddCatPI$   
 $SCInv \wedge SCAddCat \Rightarrow SCInv'$

**theorem**  $SCSetSCPI$   
 $SCInv \wedge SCSetSC \Rightarrow SCInv'$

**theorem**  $SCInterfacePI$   
 $SCInv \wedge SCInterface \Rightarrow SCInv'$

end of Z Section *sc*



# Chapter 10

## Access Control Lists

This chapter defines Access Control Lists. An ACL is a list or set of pairs of the form  $(id, perm)$  where  $id$  is a user or group identification and  $perm$  is a permission granted to  $id$ . ACL are associated with objects in the *state* section. ACL will be implemented as an abstract data type (ADT) where the hidden data structure will be an implementation of the state schema, and the interface will comprise the state operations defined below.

### 10.1 Basic Types, Parameters, and State Definition

**Z Section** *acl*, **parents:** *toolkit, main*

*GRPNAME*  $\approx$  all the possible user group names

*READ*  $\approx$  read permission

*WRITE*  $\approx$  pure write permission

*OWNER*  $\approx$  if a user has this permission, then she or he can grant all three permissions to other users or groups in the current ACL

*aclOk*  $\approx$  indicates that an ACL operation executed from a state verifying its precondition

*aclError*  $\approx$  indicates that an ACL operation executed from a state not verifying its precondition

*MAXACLEN*  $\approx$  at implementation level ACLs are of finite length, this constant defines their maximum size

$[GRPNAME]$

$SUBJECT ::= usr\langle\langle USER \rangle\rangle \mid grp\langle\langle GRPNAME \rangle\rangle$

$PERM ::= READ \mid WRITE \mid OWNER$

$ACLREPORT ::= aclOk \mid aclError$

$\mid MAXACLEN : \mathbb{N}$

We model ACLs as a relation between *SUBJECT* and *PERM*. The obvious interpretation applies: if  $(s, p) \in acl$  then user or group  $s$  has permission  $p$  in the current ACL.

<i>AccessCtrlList</i> $acl : \mathbb{F}(SUBJECT \times PERM)$
--

The invariant for this ADT is simple: the lenght of the ACL must be less or equal to the maximun allowed size.

<i>ACLInv</i> <i>AccessCtrlList</i>
$\#acl \leq MAXACLEN$

In the initial state the ACL is empty.

<i>ACLInit</i> <i>AccessCtrlList</i>
$acl = \emptyset$

## 10.2 Operations

*ACLSetMode* is the operation that sets the mode (set of permissions) for a given user or group. New permissions may be added to an ACL provided its new lenght does not go beyond the maximun. The operation is divided into two cases: *ACLSetModeOk1* to set users modes, and *ACLSetModeOk2* to set groups modes.

<i>ACLSetModeUsrOk</i> $\Delta AccessCtrlList$ $u? : USER$ $P? : \mathbb{F} PERM$ $rep! : ACLREPORT$
$\#acl - \#(acl \setminus \{usr\ u?\}) + \#P? \leq MAXACLEN$ $acl' = acl \oplus \{p : PERM \mid p \in P? \bullet usr\ u? \mapsto p\}$ $rep! = aclOk$

<i>ACLSetModeGrpOk</i> $\Delta AccessCtrlList$ $g? : GRPNAME$ $P? : \mathbb{P} PERM$ $rep! : ACLREPORT$
$\#acl - \#(acl \setminus \{grp\ g?\}) + \#P? \leq MAXACLEN$ $acl' = acl \oplus \{p : PERM \mid p \in P? \bullet grp\ g? \mapsto p\}$ $rep! = aclOk$

Now we describe the schemas for the error cases (i.e. when preconditions are not satisfied). There is just one precondition in each successful case, thus there are two error schemas.

$ACLSetModeUsrE$ $\Xi AccessCtrlList$ $u? : USER$ $P? : \mathbb{P} PERM$ $rep! : ACLREPORT$
$\#acl - \#(acl(\{usr u?\} \emptyset)) + \#P? > MAXACLEN$ $rep! = aclError$

$ACLSetModeGrpE$ $\Xi AccessCtrlList$ $g? : GRPNAME$ $P? : \mathbb{P} PERM$ $rep! : ACLREPORT$
$\#acl - \#(acl(\{grp g?\} \emptyset)) + \#P? > MAXACLEN$ $rep! = aclError$

$ACLSetMode$  is defined as the disjunction of the successful and unsuccessful cases.

$$ACLSetModeUsr \hat{=} ACLSetModeUsrOk \vee ACLSetModeUsrE$$

$$ACLSetModeGrp \hat{=} ACLSetModeGrpOk \vee ACLSetModeGrpE$$

$$ACLSetMode \hat{=} ACLSetModeUsr \vee ACLSetModeGrp$$

Now we define the operation that returns the mode of a given user or group.

$ACLGetModeOk1$ $\Xi AccessCtrlList$ $u? : USER$ $P! : \text{seq } PERM$
$\text{ran } P! = acl(\{usr u?\} \emptyset)$ $\#P! = \#(acl(\{usr u?\} \emptyset))$

$ACLGetModeOk2$ $\Xi AccessCtrlList$ $g? : GRPNAME$ $P! : \text{seq } PERM$
$\text{ran } P! = acl(\{grp g?\} \emptyset)$ $\#P! = \#(acl(\{grp g?\} \emptyset))$

$$ACLGetMode \hat{=} ACLGetModeOk1 \vee ACLGetModeOk2$$

$ACLGetSize$  is the operation that returns the current length or size of an ACL.

$ACLGetSize$ $\Xi AccessCtrlList$ $size! : \mathbb{Z}$
$size! = \#acl$

The following operation specify if a user or group has a given mode.

$\frac{\begin{array}{l} ACLIsPermOk1 \\ AccessCtrlList \\ u? : USER \\ p? : PERM \end{array}}{p? \in acl(\{usr\ u?\})}$
---

$\frac{\begin{array}{l} ACLIsPermOk2 \\ AccessCtrlList \\ g? : GRPNAME \\ p? : PERM \end{array}}{p? \in acl(\{grp\ g?\})}$
--

$$ACLIsPerm \hat{=} ACLIsPermOk1 \vee ACLIsPermOk2$$

The ADT interface is summarized.

$$ACLInterface \hat{=} ACLSetMode \vee ACLGetMode \vee ACLGetSize \vee ACLIsPerm$$

### 10.3 Proof Obligations

**theorem** ACLSetModePI

$$ACLInv \wedge ACLSetMode \Rightarrow ACLInv'$$

**theorem** ACLInterfacePI

$$ACLInv \wedge ACLInterface \Rightarrow ACLInv'$$

**end of Z Section** *acl*

## Chapter 11

# Types, Parameters, and Schemas Used to Specify Operations

In this chapter we gathered some types and schemas that are used in the definition of several operations.

### 11.1 Basic Types

#### 11.1.1 Error Reports

**Z Section** *definitions*, **parents:** *state*

The following labels are used in many operations to signal error conditions.

```
SFSREPORT ::=  
  ok  
  | userDoesNotExist  
  | objectDoesNotExist  
  | objectAlreadyExists  
  | objectIsNotOpenForReading  
  | objectIsNotOpenForWriting  
  | objectIsNotOpen  
  | permissionDenied  
  | terminalAlreadyInUse  
  | terminalNotInUse  
  | processDoesNotExist  
  | noInput  
  | noOutput
```

#### 11.1.2 Basic Modes

Files can be opened in two modes:

*read*  $\approx$  is pure read, that is the process can read from anywhere in the file but cannot modify it in any way

*write*  $\approx$  is pure write, that is the process can modify it anywhere but cannot see nothing of it

$MODE ::= read \mid write$

If a process needs to edit a file then it should open it in both modes.

## 11.2 Global Parameters

$primaryGrp\ u \approx$  is the primary group of user  $u$ ; the primary group of a user is used to set the group of a file or directory when it is created

$$\mid primaryGrp : USER \rightarrow GRPNAME$$

$write\_inode \approx$  represents the actual write of bytes into the file. Its first argument is intended to be the file where (part of) the second argument is to be written. We left it underspecified.

$read\_inode \approx$  represents the actual read of bytes from a file. We left it underspecified.

$put\_in\_mem \approx$  represents which part of a given file es mapped onto memory (cf. *Mmap*, section 4.9, and *mmap*). We left it underspecified.

$writelt \approx$  represents the actual write of bytes into a logical terminal. It must be interpreted like if part of its argument (and not necessary all of it) is written.

$$write\_inode : seq\ CCHAR \rightarrow seq\ CCHAR \rightarrow seq\ CCHAR$$

$$read\_inode : seq\ CCHAR \rightarrow seq\ CCHAR$$

$$put\_in\_mem : seq\ CCHAR \rightarrow seq\ CCHAR$$

$$writelt : seq\ CCHAR \rightarrow seq\ CCHAR$$

$$\forall F, M : seq\ CCHAR \bullet (\exists f, m : seq\ CCHAR \mid f \text{ in } F \wedge m \text{ in } M \bullet f \frown m = write\_inode\ F\ M)$$

$$\forall F : seq\ CCHAR \bullet read\_inode\ F \text{ in } F$$

$$\forall F : seq\ CCHAR \bullet put\_in\_mem\ F \text{ in } F$$

$$\forall F : seq\ CCHAR \bullet writelt\ F \text{ in } F$$

$rootdir \approx$  represents the root of the file system hierachy

$parentDir\ o \approx$  is the parent directory of  $o$

$$rootdir : OBJECT$$

$$parentDir : OBJECT \rightarrow OBJECT$$

$$parentDir\ rootdir = rootdir$$

$$\forall o : OBJECT \mid o \neq rootdir \bullet parentDir\ o \neq o$$

$suidto\ o \approx$  the user to whom a process (which was created by running program  $o$ ) can set its identity by invoking *Setuid*; objects that are not programs or programs that do not have their SUID bits on, are mapped by *suidto* to some default, non existent user; in other words this function represents a combination of the state of the SUID bit of each file and its owner

$$\mid suidto : OBJECT \rightarrow USER$$

## 11.3 Schemas Used to Specify Operations

### 11.3.1 DAC Preconditions

$$ACLI sPermForUser \triangleq ACLIsPermOk1$$

$$ACLI sPermForGrp \triangleq ACLIsPermOk2$$

The following two schemas can be used to see whether a given user has a given right over a given object. The variables used to read the right and the user are hidden in schemas *ACLI sPerForUser* or *ACLI sPermForGrp*, respectively.

$PreDACUser$
$ProcessList$ $FileSystemObjects$ $ACLI sPermForUser$ $pid? : PROCID$ $o? : OBJECT$
$pid? \in \text{dom } aprocs$ $u? = (aprocs \text{ } pid?).suid$ $o? \in objs$ $oacl \text{ } o? = \theta AccessCtrlList$

$PreDACGrp$
$Users$ $ProcessList$ $FileSystemObjects$ $ACLI sPermForGrp$ $pid? : PROCID$ $o? : OBJECT$
$pid? \in \text{dom } aprocs$ $o? \in objs$ $oacl \text{ } o? = \theta AccessCtrlList$ $\exists g : GRPNAME \mid (aprocs \text{ } pid?).suid \in grps \text{ } g \bullet g = g?$

*PreDACRead* determines whether user  $u?$  has *READ* permission in the ACL of object  $o?$ . The same, but with *WRITE*, says *PreDACWrite*. Note that both predicates check if  $u?$  has the appropriate permission through one of his groups.

$$PreDAC \triangleq PreDACUser \vee PreDACGrp \setminus (g?, u?)$$

$$PreDACRead \triangleq [PreDAC \mid p? = READ] \setminus (p?)$$

$$PreDACWrite \triangleq [PreDAC \mid p? = WRITE] \setminus (p?)$$

$$PreDACOwn \triangleq [PreDAC \mid p? = OWNER] \setminus (p?)$$

### 11.3.2 Opening an object at the *Process* level

$POpenOk1$ $\Delta Process$ $o? : OBJECT$
$or' = or \cup \{o?\}$ $ow' = ow$ $mmfr' = mmfr$ $mmfw' = mmfw$ $supr' = supr$ $mem' = mem$ $suid' = suid$ $lt' = lt$ $prog' = prog$

$POpenOk2$ $\Delta Process$ $o? : OBJECT$
$or' = or$ $ow' = ow \cup \{o?\}$ $mmfr' = mmfr$ $mmfw' = mmfw$ $supr' = supr$ $mem' = mem$ $suid' = suid$ $lt' = lt$ $prog' = prog$

$$POpenRead \hat{=} POpenOk1$$

$$POpenWrite \hat{=} POpenOk2$$

$$POpenOk \hat{=} POpenOk1 \vee POpenOk2$$

$$POpen \hat{=} POpenOk$$

$PRead$ $\Delta Process$ $sc : SecClass$ $buff : seq CCHAR$
$supr' = Sup\ supr\ sc$ $mem' = mem \frown buff$ $usr' = usr$ $or' = or$ $ow' = ow$ $mmfr' = mmfr$ $mmfw' = mmfw$ $suid' = suid$ $lt' = lt$ $prog' = prog$



<i>PAddToMem</i>
$\Delta Process$
$buff : \text{seq } CCHAR$
$mem' = mem \frown buff$ $supr' = supr$ $usr' = usr$ $or' = or$ $ow' = ow$ $mmfr' = mmfr$ $mmfw' = mmfw$ $suid' = suid$ $lt' = lt$ $prog' = prog$

### 11.3.3 Common Errors

<i>PidNotExist</i>
$\exists SecureFileSystem$
$pid? : PROCID$
$repo! : SFSREPORT$
$pid? \notin \text{dom } aprocs$ $repo! = processDoesNotExist$

<i>ObjectNotExist</i>
$\exists SecureFileSystem$
$o? : OBJECT$
$repo! : SFSREPORT$
$o? \notin objs$ $repo! = objectDoesNotExist$

<i>ObjectAlreadyExists</i>
$\exists SecureFileSystem$
$o? : OBJECT$
$repo! : SFSREPORT$
$o? \in objs$ $repo! = objectAlreadyExists$

<i>UserNotExist</i>
$\exists SecureFileSystem$
$u? : USER$
$repo! : SFSREPORT$
$u? \notin users$ $repo! = userDoesNotExist$

$\begin{array}{l} \textit{TerminalNotUsed} \\ \Xi \textit{SecureFileSystem} \\ pt? : PTERM \\ rep_0! : SFSREPORT \end{array}$
$\begin{array}{l} pt? \notin \textit{ran upt} \\ rep_0! = \textit{terminalNotInUse} \end{array}$

The following schema captures the lack of DAC permissions of a process over an object. As you can see, we do not use *AccessCtrlList*'s interface to model this error, but at implementation level programmers must use that interface.

$\begin{array}{l} \textit{PermissionDenied} \\ \Xi \textit{SecureFileSystem} \\ pid? : PROCID \\ o? : OBJECT \\ m? : MODE \\ p : PERM \\ rep_0! : SFSREPORT \end{array}$
$\begin{array}{l} o? \in \textit{objs} \\ pid? \in \textit{dom aprocs} \\ (p \notin (\textit{oacl } o?).\textit{acl}(\{ \textit{usr } (\textit{aprocs } pid?).\textit{suid} \} \})) \\ \wedge \neg (\exists g : \textit{GRPNAME} \mid g \in \textit{dom grps} \bullet \\ \quad (\textit{aprocs } pid?).\textit{suid} \in (\textit{grps } g) \\ \quad \wedge p \in (\textit{oacl } o?).\textit{acl}(\{ \textit{grp } g \} \))) \\ rep_0! = \textit{permissionDenied} \end{array}$

$$\textit{NoRead} \triangleq [\textit{PermissionDenied} \mid m? = \textit{read} \wedge p = \textit{READ}]$$

$$\textit{NoWrite} \triangleq [\textit{PermissionDenied} \mid m? = \textit{write} \wedge p = \textit{WRITE}]$$

$$\textit{NoOwner} \triangleq [\textit{PermissionDenied} \mid p = \textit{OWNER}] \setminus (m?)$$

$\begin{array}{l} \textit{MLSViolation} \\ \Xi \textit{SecureFileSystem} \\ pid? : PROCID \\ o? : OBJECT \\ rep_0! : SFSREPORT \end{array}$
$\begin{array}{l} \neg \textit{osc } (\textit{parentDir } o?) \succeq (\textit{aprocs } pid?).\textit{supr} \\ o? \notin \textit{objs} \\ rep_0! = \textit{permissionDenied} \end{array}$

**end of Z Section** *definitions*

## Chapter 12

# Main

This section was introduced just because Z/EVES does not accept the same type defined in two different sections. Otherwise *USER* would have been defined in chapters [8](#) and [10](#).

### **Z Section *main* (no parents)**

*USER*  $\approx$  all the possible users of the system

[*USER*]

**end of Z Section *main***

# Index

$\succeq$  83  
*AccessCtrlList* 88  
*ACLGetMode* 89  
*ACLGetModeOk1* 89  
*ACLGetModeOk2* 89  
*ACLGetSize* 90  
*ACLInit* 88  
*ACLInterface* 90  
*ACLInterfacePI* 90  
*ACLInv* 88  
*ACLIsPermForGrp* 93  
*ACLIsPermForUser* 93  
*ACLIsPerm* 90  
*ACLIsPermOk1* 90  
*ACLIsPermOk2* 90  
*ACLREPORT* 87  
*ACLSetModeGrpE* 89  
*ACLSetModeGrp* 89  
*ACLSetModeGrpOk* 88  
*ACLsetMode* 89  
*ACLSetModePI* 90  
*ACLSetModeUsrE* 89  
*ACLSetModeUsr* 89  
*ACLSetModeUsrOk* 88  
*ACLSetOGO* 36  
*ACLToSeqCHAR* 57  
*allgrp* 18  
*aprocs* 21  
*CATEGORY* 82  
*CCHAR* 14  
*CHAR* 14  
*ChinscE1* 25  
*ChinscE2* 25  
*ChinscE* 25  
*Chinsc* 25  
*ChinscOk* 25  
*ChobjscE1* 33  
*ChobjscE2* 33  
*ChobjscE3* 33  
*ChobjscE* 33  
*Chobjsc* 33  
*ChobjscOk1* 32  
*ChobjscOk2* 32  
*ChobjscOk* 33  
*ChsubscE1* 34  
*ChsubscE2* 34  
*ChsubscE3* 34  
*ChsubscE* 34  
*Chsubsc* 34  
*ChsubscOk* 34  
*CloseE1* 36  
*Close* 36  
*CloseOk* 35  
*cptcs* 15  
*CreateE1* 38  
*CreateE2* 38  
*CreateE3* 38  
*CreateE* 38  
*Create* 38  
*CreateOk1* 37  
*CreateOk2* 38  
*CreateOk* 38  
*ExecE1* 40  
*ExecE2* 40  
*ExecE3* 40  
*ExecE* 40  
*Exec* 40  
*ExecOk* 39  
*FileSystemObject* 19  
*ForkE* 41  
*Fork* 41  
*ForkOk* 41  
*FSOInit* 19  
*FSOInv* 72  
*FSOInvIF* 72  
*FSOInvWF* 71  
*GetE* 66  
*Get* 66  
*GetOk* 66  
*GRPNAME* 87  
*GRPP* 36  
*grps* 18  
*Inf* 83  
*INF* 83

*InputE* 27  
*input* 14  
*Input* 27  
*InputOk* 27  
*LCONT* 16  
*L* 83  
*LinkE3* 42, 53  
*LinkSE3* 44  
*LinkE1* 42  
*LinkE2* 42  
*LinkE4* 42  
*LinkE* 42  
*Link* 42  
*LinkOk* 42  
*LinkSE1* 44  
*LinkSE2* 44  
*LinkSE4* 44  
*LinkSE* 44  
*LinkS* 44  
*LinkSOk* 43  
*LogicalTerminals* 16  
*LoginE1* 29  
*LoginE2* 29  
*LoginE* 29  
*Login* 29  
*LoginOk* 28  
*LTCAdd* 66  
*LTCInit* 16  
*ltcont* 16  
*LTCRead* 51  
*LTCRemove* 67  
*LTERM* 16  
*lt* 20  
*LTIInit* 16  
*LTIInv* 71  
*ltsc* 16  
*MAX\_ACL\_LEN* 87  
*MAXLEVEL* 82  
*MAXNCAT* 82  
*mem* 20  
*MLSViolation* 96  
*MmapE1* 46  
*MmapE2* 46  
*MmapE3* 46  
*MmapE* 46  
*Mmap* 46  
*MmapOk1* 45  
*MmapOk2* 46  
*MmapOk* 46  
*mmfr* 20  
*mmfw* 20  
*MODE* 91  
*mptsc* 14  
*NoOwner* 96  
*NoRead* 96  
*NoWrite* 96  
*null* 14  
*oacl* 19  
*ObjectAlreadyExists* 95  
*OBJECT* 19  
*OBJECTToSeqCHAR* 43  
*objs* 19  
*ocont* 19  
*OCONT* 19  
*OpenE1* 48  
*OpenE2* 48  
*OpenE31* 48  
*OpenE32* 48  
*OpenE* 48  
*OpenFrame* 47  
*Open* 48  
*OpenOk1* 47  
*OpenOk2* 47  
*OpenOk* 47  
*or* 20  
*osc* 19  
*OscstatE1* 48  
*OscstatE2* 48  
*OscstatE* 48  
*Oscstat* 48  
*OscstatOk* 48  
*OTHP* 36  
*output* 14  
*ow* 20  
*OWNP* 36  
*PAddToMem* 95  
*parentDir* 92  
*PClose* 35  
*PERM* 87  
*PermissionDenied* 96  
*PExec* 39  
*PhysicalTerminal* 14  
*PidNotExist* 95  
*PInit* 21  
*PInv* 72  
*PLInit* 21  
*PLogin* 28  
*PMmap* 44  
*POpen* 94  
*POpenOk1* 94

*POpenOk2* 94  
*POpenOk* 94  
*POpenRead* 94  
*POpenWrite* 94  
*PRead* 95  
*PreDACGrp* 93  
*PreDAC* 93  
*PreDACOwn* 93  
*PreDACRead* 93  
*PreDACUser* 93  
*PreDACWrite* 93  
*primaryGrp* 92  
*Process* 21  
*ProcessList* 21  
*PROCID* 21  
*prog* 21  
*PSetuid* 54  
*PTGet* 65  
*PTInit* 14  
*PTInput* 26  
*PTPut* 67  
*pts* 14  
*PutE1* 68  
*PutE2* 68  
*PutE* 68  
*Put* 68  
*put\_in\_mem* 92  
*PutOk* 68  
*ReadE1* 50  
*ReadE2* 50  
*ReadE* 50  
*Read* 50  
*read\_inode* 92  
*ReadLTE1* 52  
*ReadLTE2* 52  
*ReadLTE* 52  
*ReadLT* 52  
*ReadLTOk* 52  
*ReadOk* 50  
*ready* 14  
*RenameE1* 53  
*RenameE2* 53  
*RenameE4* 53  
*RenameE* 53  
*Rename* 53  
*RenameOk* 53  
*rootdir* 92  
*rootgrp* 18  
*root* 18  
*SCAddCatE1* 85  
*SCAddCatE2* 85  
*SCAddCatE* 85  
*SCAddCat* 85  
*SCAddCatOk* 84  
*SCAddCatPI* 86  
*SCGetCat* 84  
*SCGetLevel* 84  
*SCGetSize* 84  
*SCInit* 83  
*SCInterface* 86  
*SCInterfacePI* 86  
*SCInv* 83  
*SCREPORT* 82  
*SCSetLevelE* 85  
*SCSetLevel* 85  
*SCSetLevelOk* 85  
*SCSetLevelPI* 86  
*SCSetSCE1* 86  
*SCSetSCE2* 86  
*SCSetSCE* 86  
*SCSetSC* 86  
*SCSetSCOk* 86  
*SCSetSCPI* 86  
*SCToSeqCHAR* 48  
*secadm* 18, 77  
*SECADMIN* 77  
*SecClass* 82  
*SecClassToUserSecClassD* 78  
*SecClassToUserSecClassX* 80  
*SecureFileSystem* 22  
*SetuidOk1* 55  
*SetuidOk2* 56  
*SetuidOk3* 56  
*SFSInit* 22  
*SFSInv* 73  
*SFSInvPLTI* 73  
*SFSInvPLTO* 73  
*SFSInvUT* 73  
*SFSInvWF* 72  
*SFSREPORT* 91  
*softtcb* 19  
*StatOk* 57  
*stdin* 15  
*stdout* 15  
*SUBJECT* 87  
*suid* 20  
*suidto* 92  
*Sup* 83  
*SUP* 83  
*supr* 20

*TERM* 14  
*TerminalNotUsed* 96  
*ttys* 16  
*UInit1* 18  
*UInit* 18  
*UInv* 71  
*UInvWF* 71  
*upt* 15  
*USCAddCatE1* 79  
*USCAddCatE2* 79  
*USCAddCatE3* 79  
*USCAddCatE* 79  
*USCAddCat* 79  
*USCAddCatOk* 79  
*USCAddCatPI* 81  
*USCErrorReport* 78  
*USCGetCatE* 80  
*USCGetCat* 80  
*USCGetCatOk* 80  
*USCGetLevelE* 80  
*USCGetLevel* 80  
*USCGetLevelOk* 80  
*USCGetSizeE* 80  
*USCGetSize* 80  
*USCGetSizeOk* 80  
*USCInit* 78  
*USCInterface* 80  
*USCInterfacePI* 81  
*USCInv* 78  
*USCREPORT* 77  
*USCSetLevelE1* 79  
*USCSetLevelE2* 79  
*USCSetLevelE* 79  
*USCSetLevel* 79  
*USCSetLevelOk* 79  
*USCSetLevelPI* 81  
*USCSetSCE1* 79, 80  
*USCSetSCE3* 80  
*USCSetSCE* 80  
*USCSetSC* 80  
*USCSetSCOk* 79  
*USCSetSCPI* 81  
*USCUserNotExist* 78  
*USER* 97  
*UserNotExist* 96  
*UsersAndTerminals* 15  
*UserSecClass* 77  
*users* 18  
*Users* 18  
*usr* 20  
*UInit* 15  
*UInvCSC* 70  
*UInv* 70  
*UInvI* 70  
*UInvO* 70  
*WriteE1* 60  
*WriteE2* 60  
*WriteE3* 60  
*WriteE* 60  
*Write* 60  
*write\_inode* 92  
*WriteLTE1* 63  
*WriteLTE2* 63  
*WriteLTE* 63  
*WriteLT* 63  
*writel* 92  
*WriteLTOk* 62  
*WriteOk1* 59  
*WriteOk2* 59  
*WriteOk* 60

# Bibliography

- [1] ABRAMS, M. D., JAJODIA, S., AND PODELL, H. J. *Information Security: an integrated collections of essays*. IEEE Computer Society press, 1995.
- [2] BELL, D. E., AND LAPADULA, L. Secure computer systems: Mathematical foundations. *Technical Report MTR-2547 I-III* (Dec. 1973).
- [3] BELL, D. E., AND LAPADULA, L. Secure computer systems: Mathematical model. *Technical Report ESD-TR-73-278 II* (Nov. 1973).
- [4] CLARKE, D. D., AND WILSON, D. R. A comparison of commercial and military computer security policies. In *IEEE symposium on security and privacy* (1987), pp. 184–194. IEEE Computer Society Press.
- [5] CRISTIÁ, M. Formal verification of an extension of a secure, compatible UNIX file system. Master’s thesis, Instituto de Computación, Universidad de la República, Uruguay, <http://www.fceia.unr.edu.ar/gidis>, 2002.
- [6] CRISTIÁ, M., GIUSTI, G., AND MANZANO, F. *Guía del Diseño y la Implementación de GTL 0.1*. GIDIS, [www.fceia.unr.edu.ar/gidis](http://www.fceia.unr.edu.ar/gidis), July 2003.
- [7] DENNING, D. E. A lattice model of secure information flow. *Communications of the ACM* 19, 5 (May 1976), 236–243.
- [8] EVANS, A. Specifying and verifying concurrent systems using Z. In *FME ’94: Industrial Benefit of Formal Methods* (1994), Springer-Verlag, pp. 366–380.
- [9] GASSER, M. *Building a Secure Computer System*. Van Nostrand Reinhold, 1988.
- [10] JACKY, J. *The Way of Z*. Cambridge University Press, 1997.
- [11] LANDWEHR, C. E. Formal models for computer security. *ACM Computing Surveys* 13, 3 (Sept. 1981), 247–278.
- [12] POTTER, B., SINCLAIR, J., AND TILL, D. *An Introduction to Formal Specification and Z*. Prentice Hall International, 1996.
- [13] ZAVE, P., AND JACKSON, M. Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology* 6, 1 (Jan. 1997).