Formal Specification of Operating System Operations

Dan Zhou
Florida Atlantic University
dan@cse.fau.edu

Paul E. Black
National Institute of Standards and Technology
paul.black@nist.gov

Abstract

This paper describes the development of a formal specification of a secure operating system architecture and its security-related operations in higher-order logic theorem prover, HOL. We specify operating system entities as user defined data types and security conditions as predicates that restricts system operations. We also provide a uniform environment for system commands that change security state of systems.

1. Motivation

An operating system (OS) is an important piece of the puzzle that provides system security as it enforces security policies of applications. The correctness of the operating system itself is paramount in ensuring the security of applications.

To facilitate high-assurance system development, we formally specify operating system security policies in higher-order logic theorem prover, HOL [4]. Formal specifications enables us to describe OS policies clearly and precisely. It also could benefit us in two more ways: (1) formal analysis using HOL theorem proving capability [8], (2) automated test generation based on formal specifications. In a related work, we are investigating methods of automatically generating tests from HOL specifications.

Some previous works focus on abstract models of security protocols, which abstract away any detail that pertains to a real system requirement (e.g., [5]); this type of specification and analysis is useful to protocol designers. Others have specified high-level security policy models that can be used to validate real systems without getting into the detail making of actual systems [6, 3].

In this work we model OS at a level that is close to system requirement which makes the model useful for system development. An OS at this level is large and complex, attributes which are major obstacles in specification effort. We overcome this obstacle partly by borrowing object modeling technologies: dividing security policy into data types, functions on these data types, and operations and constraints on these operations that move the system from state to state.

We specify concrete entities that are components comprising the objects and subjects of operating systems. We also specify security conditions for system operations. We then describe security-related operations which present a user view's of operating systems. Specifically we define different types of entities in OS as HOL types and define security checks on these entities as HOL functions. We then present a uniform view of UNIX shell commands by providing an operating environment as lists of binding of variable names, types, and values.

The specification of secure operating system is a part of much larger effort in the late-development testing and post-development accreditation process. The specification developed in this work will be validated vigorously against a higher-level security policy model and will be used for automated test generation.

In the rest of the paper, we first describe entities in OS in Section 2, we then describe security conditions for system operations in Section 3. In Section 4 we specify system states and the state transitions. We summarize in Section 5.

2. Operating System Entities

The subject of our case study is PitBull [1], a security product that enhances the security of Solaris 7. It implements a more restricted version of the Bell-LaPadula (BLP) model.

A security policy defines the rules that govern the operations of systems. A policy model provides a framework for defining rules. The BLP model addresses the security concerns of multi-level information systems [2]. Abstract entities in the model are subjects and objects. Objects model resources to be controlled. Subjects are active entities that seek access, such as *read* or *write*, to objects.

The BLP model prevents unauthorized access to information through mandatory access control (MAC) mechanism, where the access control is mandated by the system. Subjects and objects are classified according to clearance and sensitivity level, respectively. They are also assigned compartments to which they belong. Subjects are granted access rights to objects based on their clearance and need-to-know categories.

In addition to MAC, PITBULL also provides standard UNIX discretionary access control (DAC), where owners of information can restrict or grant access to the information using permission bits.

The only subjects in an OS are processes: there are normal processes and system processes. There are many different types of objects in an OS, such as regular file system objects (files, directories, etc), processes, and X-window objects. In this work we define subjects and objects by their security-related attributes: DAC attributes, MAC attributes, identifications and privileges. PITBULL relies on security-related attributes of subjects, objects, and the system to grant or deny access rights of subjects to objects.

2.1. DAC Attributes

Each object in UNIX has a set of permission bits, one set per type of users. We model the permission bits for each type of user as a record type *PBS*

```
PBS = <|r: bool; w: bool; x: bool|>
```

HOL record type *DAC*

```
DAC = <| up: PBS; gp: PBS; op: PBS|>
```

represents the set of permission bits for users, group and the world. In HOL definition, `<| ... ... |>` represents a record and ";" is a field delimiter.

2.2. MAC Attributes

Among the MAC attributes classifications and clearances are hierarchical while compartments not.

MAC Labels PitBull provides four types of MAC labels: sensitivity label (SL), clearance label (CL), information label (IL) and integrity label (TL). SLs and CLs consist of classification and compartment. SL is used for access control, CL is used to control operations on MAC labels. HOL record type *SCLabel* represents both SLs and CLs:

```
SCLabel = <|class: Class; comp: Comp set |>
```

where type *Class* represents classifications and type *Comp* represents compartments. The postfix construct *set* is a type constructor. We define HOL types *ILabel* and *TLabel* for IL and TL similarly.

Classifications and Compartments Site system administrators can set values for classifications and compartments. As an example we define *Class* for classification as an enumeration type with values *classTS*, *classS*, *classU*:

```
Class =  classTS | classS | classU
```

Functions *classGT* and *classGTE* define *greater than* and *greater than or equal to* relationship among classifications.

We define *Comp* as an enumeration type with four values *NIST*, *ITL*, *FAU* and *CSE* for compartments:

```
Comp = NIST | ITL | FAU | CSE
```

HOL function *RLLDOM* defines a dominance relation between any two MAC labels. Function *RSLSL* defines the dominance relation between two SLs.

2.3. Privileges

The BLP model disallows write-down operations, some of which are necessary for day-to-day operation of computer systems. Privileges of subjects are means to bypass restrictions the BLP model imposes and to perform otherwise unauthorized operations. A subjects can obtain privileges by executing an executable file that contains special privileges. We describe privilege in detail to illustrate the complexity of OS security policy.

There are three groups of privileges implemented on the PITBULL system: general privileges, X-Window privileges, and superuser privileges (denoted by *Priv*, *XPriv*, and *SUPriv*, respectively). The general privileges are organized into eight functionality sub-groups (Table 1). Table 1 lists the number of privileges in each sub-group. Each sub-group contains privileges related to a particular aspect of system security. We define one data type in HOL for each group to denote the group specific privileges. A general privilege is either one of the privileges listed in Table 1 or PV_ROOT. A privilege is either a general privilege, or an X-Window privilege, or a superuser privilege.

Privileges are organized hierarchically as a forest with each group as a rooted tree (Figure 1). A privilege of higher level in the hierarchy contains all the privileges that are lower in the hierarchy.

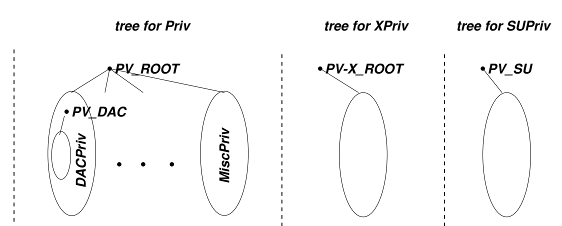

Figure 1. Privilege Hierarchy

To check if a subject has a particular privilege *pv*, we need to find out if *pv* or a privilege higher in hierarchy than

Table 1. General System Privileges

Name	Description	No. of Privileges
AUPriv	Audit privileges allowing operations related to audit system	7
AZPriv	Authorization privileges allowing operations related to process authorization	4
DACPriv	DAC privileges allowing processes to bypass DAC-related restrictions	12
FSPriv	File system privileges related to file systems	2
LABELPriv	Label privileges related to access of labels such as information label and sensitivity label	17
MACPriv	MAC privileges allowing processes to bypass MAC restrictions	13
ASNPriv	Network, driver, and STREAM privileges	5
PVPriv	Privileges allowing processes to modify the privilege sets of files or processes	2
SRPriv	Privileges related to all other types of system resource	6
MiscPriv	Miscellaneous privileges	7

pv (i.e., an ancestor privilege) is in the subject's privilege set. Two functions are defined to help this checking: a parent function that finds parent privilege given a particular privilege, and a privilege checking function that recursively checks if a privilege or any of its ancestors is in a given privilege set.

3. Security Conditions

Security conditions are the rules that system operations obey. They can be rather complex. Sometimes they are also confusing because of the many factors contributing to a decision making process. The complexity of the conditions warrants formal specification and analysis to ensure quality of the systems.

As an example we show the condition for modifying object's sensitivity level (SL). A process can change the SL of an object or subject if and only if

- The process has either privilege PV_SL_FILE (object only) or PV_SL_PROC (subject only)
- The process has DAC WRITE access to the object (object only)
- The process has MAC WRITE access to the object or subject
- The process has DAC OWNER access to the object or subject (downgrade only)
- The process has either privilege PV_SL_UG (upgrade only) or PV_SL_DG (downgrade only)
- The process has privilege PV_MAC_CL (if either the new SL or the existing SL is not dominated by the acting process's CL)
- The new SL dominates the IL and is dominated by the CL (subject only).

The condition for modifying SL of an object is defined as *procMdSLCond* in HOL:

```
procMdSLCond s p os sl =
(if (isSubjObj os) then
    (hasPV_PSP s p /\
     RLLDOM (scLabel sl)
            (iLabel (ilMACObj os)) /\
     RLLDOM (scLabel (clMACObj os))
            (scLabel sl))
 else (hasPV_PSF s p /\ hasRT_DW p os))  /\
hasRT_MW p os      /\
(if ~(RSLSLDOM (slMACObj os) sl)
 then (hasRT_DO p os /\ hasPV_PSD s p)
 else (hasPV_PSU s p))      /\
(if (~(RSLSLDOM p.MAC.CL sl) \/
   ~(RSLSLDOM p.MAC.CL (slMACObj os)))
 then hasPV_PMC s p
 else T)
```

4. System Operations

User commands typed at a UNIX prompt are executed by system processes. Execution of these commands move the OS from state to state.

4.1. System States

A system state is the collection of the entities that are active in OS and the values of these entities. A state is represented as a list of entity names bound with their values.

We use a pair to represent the binding of the name and value of an entity. Names and values are represented uni-

Table 2. General System Privileges

name	description	definition
varE	return an entity given a name	`varE (h::t) x = if (varB h = x) then valB h else varE t x`
unpkSys	return a system given a value	`unpkSys (sysVal s) = s`
unpkFile	return a file given a value	`unpkFile (fileVal f) = f`
unpkProc	return a process given a value	`unpkProc (subjVal p) = p`
sysE	return a system from a store	`sysE st s = unpkSys (varE st (sysVar s))`
fileE	return a file from a store	`fileE st f = unpkFile (varE st (fileVar f))`
procE	return a process from a store	`procE st p = unpkProc (varE st (subjVar p))`

formly as unions containing the type information of the actual entity. Functions *varB* and *valB* retrieves the name and value of a given entity.

We define a *store* as a list of bindings to represent the operating environment of an operation, or the state of the operating system. A change to the environment is represented by prepending a binding to the *store*. To evaluate a variable, we scan the *store* from the beginning until we find a match, at which point the corresponding value is returned. Table 2 lists auxiliary functions that evaluate values of entities.

4.2. State Transitions

A security policy has two components: if an operation should be permitted (authorization) and how it changes the system state if the operation is executed (state transition). Security rules determine the circumstances under which a requested should be granted or rejected.

For example, the execution of a command to set sensitivity label of a file is divided into two parts. First, authorization: Function *setSLFileCond* utilizes *procMdSLCond* to check the access right of the process making the change:

```
setSLFileCond st p f sl =
procMdSLCond (sysE st system) (procE st p)
   (fsoObj (fileFSO (fileE st f))) sl
```

where *sl* is the new sensitivity label to be set on file *f*. Second, state transition: Function *setSLFile* describes how the system state is to be changed:

```
setSLFile (f:FileType) (sl:SCLabel) =
(f with MAC := (f.MAC with SL := sl))
```

Function *setSLFileEnv* models the operation of a process attempting to set the sensitivity label of a file. This operation may or may not change the state of the system, depending on if the process has the right to perform the operation.

```
setSLFileEnv st
   [strArg p; strArg f; sclArg sl] =
if setSLFileCond st p f sl
then update st (fileVar f)
     (fileVal (setSLFile (fileE st f) sl))
else st
```

Function *update* prepends new values to the *store*. We can define *setSLProc* as a command to change process sensitivity labels similarly.

The system evolves as commands are typed at the prompt and are executed. The *nextstate* function models this evolution:

```
(nextstate st1 set_sl_file arglist1 =
   setSLFileEnv st1 arglist1) /\
(nextstate st2 set_sl_proc arglist2 =
   setSLProcEnv st2 arglist2)
```

4.3. Examples

Let's look at a UNIX command that changes file sensitivity labels. Suppose we have a file *f1* whose uid is 1 and gid is 10. Its permission bits are `rw-r--r--`. Its classification is *classTS* and its compartment set is $\{$*NIST*$\}$.

We have a process *p1* whose euid is 1 and egid is 10. It only has privilege PV_DAC. Its sensitivity label (SL) is as follows: clearance level is of *classTS* and the compartment set is $\{$*NIST*$\}$. Its clearance label (CL) is the same as its SL.

Process *p2* has same attributes as *p1*, except that it also has privileges PV_SL_FILE and PV_SL_DG.

The clearance labels that the system *system* is authorized to handle are between *classS* and *classTS*. The compartments that *system* is authorized to process are *NIST* and *FAU*.

The initial state of OS is stored as a list:

```
state1 = [(sysVar "system", sysVal system);
          (fileVar "f1", fileVal f1);
          (subjVar "p1", subjVal p1);
          (subjVar "p2", subjVal p2)]
```

Suppose *sl2* is a SL with clearance level *classS* nd compartment set {*NIST*}. A request by process *p2* to set the sensitivity label of *f1* to *sl2* is decided by predicate

```
setSLFileCond state1 "p2" "f1" sl2.
```

Process *p2* is allowed to perform the operation based on conditions listed in *setSLFileCond*. The state of OS changes to

```
state2  = [(sysVar "system", sysVal system);
          (fileVar "f1", fileVal f1new);
          (subjVar "p1", subjVal p1);
          (subjVar "p2", subjVal p2)],
```

where *f1new* is the *f1* except that its classification is *classS*.

5. Experience

We specified a security operating systems architecture. This is a complex system with the large number of types of entities and security conditions enforced on the system operations.

HOL specification language is suitable for describing large systems because of its expressibility. HOL provides a variety of ways to define data type, similar to those that can be defined using a BNF format. The ability to define types and functions recursively makes specification writing similar to that of writing in a high-level programming language. Unlike a high-level programming language, HOL is a strongly-typed language and HOL expressions are precise.

This project started after the product had been developed. It was desirable to have a formal model for accreditation purpose. Informal security policy description and security architecture were available to us. There were a few difficulties that we ran into. The main difficulty came from the complexity of the system. We spent a lot of time to familiarize ourselves with the inter-weave of requirements that appear in different places. As as specification aid, we used object-oriented modeling technology UML to help organize the formal specifications [7]. A second difficulty came from the ambiguity of informal descriptions that are available to us. We found a few inconsistency and mistakes in the published informal descriptions. On other occasions we spent significant amounts of time to explain our confusion to system developers to get a clear answer.

Formal specifications are also subject to mistakes if there are no validations. For example we once mistakenly wrote a *less than* instead of a *greater than* in a relationship definition. A formal evaluation would help as would a formal analysis of specifications. We plan to do just that in a follow up study.

We also plan to use these formal specifications to generate tests and evaluate the design against security criteria.

References

[1] Argus Systems Group, Inc. *Trusted Facility Manual, Argus Security Solutions for Solaris 7, Fortify/PitBull*, September 1999.

[2] D. Elliott Bell and Leonard J. LaPadula. Secure computer systems: Mathematical foundations. Technical Report 2547, MITRE Corporation, Bedford, MA, 1973.

[3] Anthony Boswell. Specificaiton and validation of a security policy model. *IEEE Transactions on Software Engineering*, 21(2):63–68, February 1995.

[4] M.J.C. Gordon. A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subramanyam, editors, *VLSI specification, verification and synthesis*. Kluwer, 1987.

[5] Jeremy Jacob. Security specifications. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 14–23, 1988.

[6] John McLean. The specification and modeling of computer security. *IEEE Computer*, pages 9–15, January 1990.

[7] Dan Zhou. Towards the formal modeling of a secure operating system. In *Proceedings of the 23rd National Information System Security Conference*, Baltimore, Maryland, October 2000.

[8] Dan Zhou and Shiu-Kai Chin. Formal Analysis of a Secure Communication Channel: Secure Core-Email Protocol. In *The Proceedings of World Congress on Formal Methods in the Development of Computing Systems (FM'99)*, Toulouse, France, September 1999.