Even more patterns for secure operating systems

Eduardo B. Fernandez, Tami Sorgente, and Maria M Larrondo-Petrie
Dept. of Computer Science and Engineering
Florida Atlantic University
Boca Raton, FL
{ed, tami, maria@cse.fau.edu}

Abstract

An operating system (OS) interacts with the hardware and supports the execution of all the applications. As a result, its security is very critical. Many of the reported attacks have occurred through the OS (kernel and utilities). The security of individual execution time actions such as process creation, memory protection, and the general architecture of the OS are very important and we have previously presented patterns for these functions. We present here patterns or the representation of processes and threads, emphasizing their security aspects. We also present a pattern to control the use of commands by administrators.

1 Introduction

The operating system (OS) acts as an intermediary between the user of a computer and its hardware. The purpose of an OS is to provide an environment in which users can execute programs in convenient and efficient manner [Sil05]. OSs control and coordinate the available resources to present to the user an abstract machine with convenient features. Clearly, the security of operating systems is very critical since the OS supports the execution of all the applications as well as access to persistent data.

We have presented several patterns for different aspects of the security of operating systems [Fer02, Fer03, Fer05, Fer06a, Sch06]. These are patterns intended for designers of such systems but clearly are useful for teaching security, we use these patterns in our security courses and in a coming textbook [Fer06b]. We present here security patterns for three additional aspects. We assume the reader to be familiar with basic security concepts [Fer06b, Gol06, Pfl03]. Figure 1 shows the relationships of the new patterns with respect to each other and with respect to some previously presented patterns (the patterns presented here are shown with double margins). Their thumbnail descriptions are given below, starting with the three new patterns:

Secure Process /Thread. How do we make the execution of a process secure? A process is a program in execution and the unit of execution in some operating systems. A secure process is also a unit of execution isolation as well as a holder of rights to access resources. A variant describes how to make secure the execution of a thread. A thread is a lightweight process. A secure thread is a thread with controlled access.

Virtual Address Space Structure. How do we select the virtual address space for OSs that have special security needs? Some systems emphasize isolation, others information sharing, others good performance.

Administrator Hierarchy. How do we limit access rights for administrators? Define a hierarchy of system administrators with controlled rights using a Role-Based Access Control (RBAC) model.

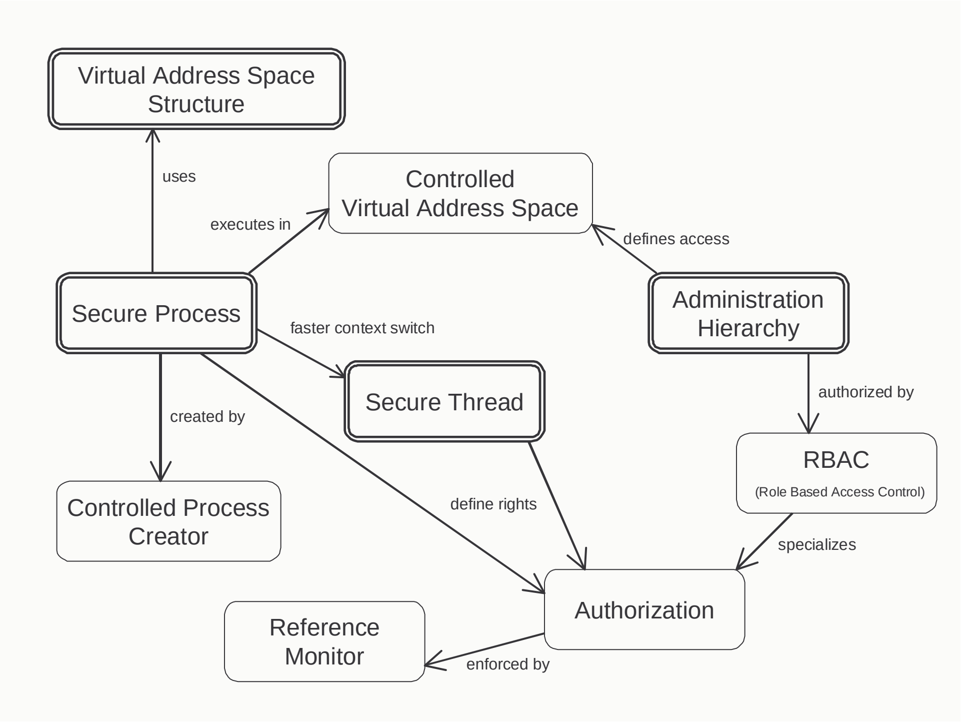


Figure 1. Pattern diagram for the patterns discussed here and their relationship to past patterns

Controlled Virtual Address Space [Fer02]. How to control access by processes to specific areas of their virtual address space (VAS) according to a set of predefined rights? Divide the VAS into segments that correspond to logical units in the programs. Use special words (*descriptors*) to represent access rights for these segments.

Controlled-Process Creator [Fer03]. How to define the rights to be given to a new process? Define rights as part of its creation. Give it a predefined subset of its parent's rights.

Authorization [Sch06]. How do we describe who is authorized to access specific resources in a system? Keep a list of authorization rules describing who has access to what and how. Authorization describes the rules of an Access matrix model.

Role-Based Access Control [Sch06]. How do we assign rights to people based on their functions or tasks? Assign people to roles and give rights to these roles so they can perform their tasks.

Reference Monitor [Sch06]. How to enforce authorizations when a process requests access to an object? Define an abstract process that intercepts all requests for resources from processes and checks them for compliance with authorizations.

Section 2 presents the Secure Process pattern and its variant the Secure Thread. The Virtual address space structure is described in Section 3, while Administrative roles are presented in Section 4.

2 Secure Process

How do we make the execution of a process secure? A process is a program in execution and the unit of execution in some operating systems. A secure process is also a unit of execution isolation as well as a holder of rights to access resources.

2.1 Example

A group of designers in Company X built an operating system and did not put any mechanisms to control the actions of processes. This resulted in processes being able to access the address space and other resources of the other processes. In this environment we cannot protect the shared information nor assure the correct execution of any process (their code and stack sections may be corrupted by other processes).

2.2 Context

Typically, OSs support a multiprogramming environment with several user-defined processes active at a given time. During execution it is essential to maintain all information regarding the executing process, including its current *status* (the value of the program counter), the contents of the processor's registers, and the process stack containing temporary data (subroutine parameters, return addresses, temporary variables, and unresolved recursive calls). All this information is called the *process context*. When a process needs to wait, the OS must save the context of the first process and load the next process for execution, this is a context switch. The saved process context is brought back when the process resumes execution.

2.3 Problem

We need to control the resources accessed by a process during its execution and protect its execution data from other processes. Proper maintenance of process execution is essential not only for context switching, but also for security in maintaining a separate structure for each execution.

The possible solution to this problem is constrained by the following forces:

- If processes have unrestricted access to resources they can interfere with the execution of other processes. We need to control what resources they can access. Processes should be given only the rights they need to perform their functions (need to know principle [Gol06, Fer06b]).
- Each process requires some data, a stack, space for temporary variables, keeping status of its devices and other information. All this information resides in its address space and needs to be protected.

2.4 Solution

Assign to each process a set of rights to access the resources they need. Assign also a unique address space to store all the data it needs during execution as well as data shared with other processes. This protects processes from interference from the other processes, assuring confidentiality and integrity of the data. In the **ProcessDescriptor,** a data structure containing all the information a process needs for its execution, add functions to make execution secure, specifically access to any resource must be explicitly authorized. It may also be possible to add resource quotas to avoid denial of service problems but this requires some global resource usage policies.

Structure

Figure 2 shows a **ProcessDescriptor** that contains the status of a process. Each **ProcessDescriptor** has a **Subject** as owner with predefined rights for specific **Resources (**these rights are defined by the Authorization pattern). More than one ProcessDescriptor can be created, corresponding to multiple executions of **ProgramCode,** and describing different processes. A unique **VirtualAddressSpace** is associated with each process (defined by the Controlled Virtual Address Space pattern). The process **Code** as well as the process Stack and any temporary data are stored in the VirtualAddressSpace of the process. The Process Descriptor defines explicit rights for the process to access resources (these rights must be a subset of the subject's rights). These accesses are enforced by the Reference Monitor pattern.

Dynamics

Figure 3 shows a sequence diagram for the use case "Access a resource". The process request is intercepted by the Reference Monitor which determines if it is authorized (checkAccess operation in the Right). If it is, the access proceeds.

2.5 Implementation

The Process Descriptor is typically called Process Control Block (PCB), or Task Control Block (TCB), and includes references (pointers) to its code section, its stack, and other needed information. There are different alternatives to implement data structures in general [Nyh05]. Records (structs in C) are typically used for the Process descriptor. The Process Descriptors of the processes in the same state are usually linked together in a double-linked list. The hardware may include registers for some of the attributes of the ProcessDescriptor. For example, the Intel X86 Series includes registers for typical attributes. There are different ways to associate virtual address space to a process [Fer06b]. There are also different ways of associating rights with a new process, see the Controlled Process Creator in [Fer03]. The hardware architecture should also restrict access to memory within the predefined ranges for each process.

The pattern models as shown describe directly models where subjects have rights such as the Access Matrix and Role-Based Access Control [Fer01, Sch06]. Some operating systems use Multilevel (typically mandatory) models where the access of a process is decided by its level with respect to the accessed resource [Sch06]. In the latter case, the process instead of being given a right has a tag or label that indicates its level. Resources have similar tags and the Reference Monitor compares both tags.

2.6 Example resolved

After adding rights to a process representation each process is constrained to access only those resources for which it has rights. This protects each process from each other as well as their virtual address spaces. The confidentiality and integrity of shared data is protected as well.

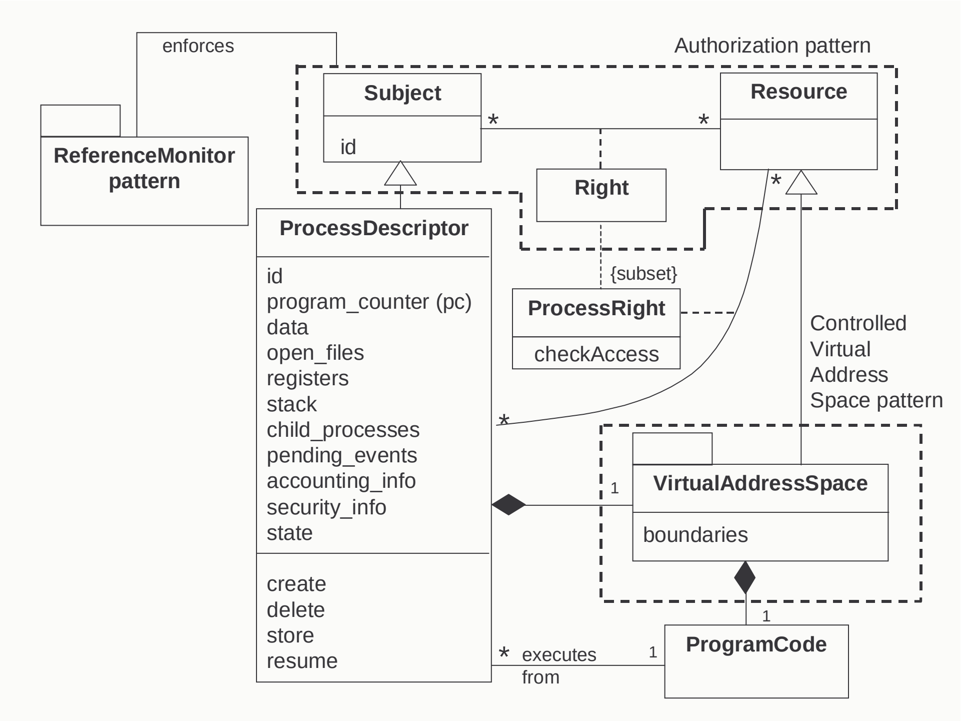


Figure 2.Class diagram for Secure Process

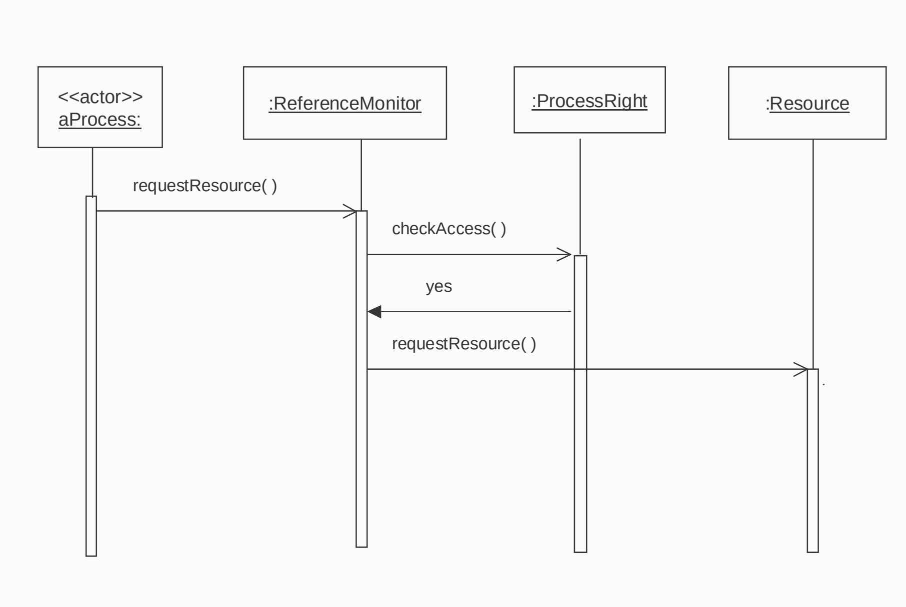


Figure 3. Sequence diagram for use case “Access a resource”.

.

2.7 Variant

Secure Thread. How do we make the execution of a thread secure? A thread is a lightweight process. A secure thread is a thread with controlled access to some resources. Figure 4 represents the addition of the **ThreadDescriptor** to the secure process. One Process may have multiple threads of execution. Each thread is represented by a **ThreadDescriptor**. A unique **VirtualAddressSpace** is associated with a process ad shared by peer threads. **ThreadRights** define access rights to the VAS.

Threat status includes typically a stack, a program counter, and some status bits. There are different ways to associate threads with a process [Sil05]. Typically, several threads are collected into a process. Threads can be created with special packages, e.g., POSIX in Unix, or through the language, as in Java or Ada. Rights can be added explicitly or we can use the hardware architecture enforcement of the proper use of the process areas (see Known Uses).

2.8 Known uses

- Linux uses records for process descriptors. One of the entries defines the process credentials that define its access to resources [Nut03, Sil05]. Other entries describe its owner (subject) and process id.
- Windows NT and 2000. Resources are defines as objects (really classes). The process id is used to decide access to objects [Sil05]. Each file object has a security descriptor which indicates the owner of the file and an access control list that describes the access rights for the processes to access the file.
- Solaris threads have controlled access to resources defined in the application, e.g. when using the POSIX library [Sil05].
- Operating systems running on Intel architectures can protect thread stacks, data, and code by placing them in special segments of the shared address space (with hardware-controlled access).

2.9 Consequences

This pattern has the following advantages:.

- It is possible to give specific rights for resources to each process which makes it possible to apply the least privilege principle.
- The process' resources can be protected from other processes, because they are restricted to access only authorized resources.

This pattern has the following disadvantage:

- There is some overhead in using a Reference Monitor to enforce accesses.
- It may not be clear what rights to assign toeach process.

2.10 Related patterns

- Controlled Process Creator [Fer03]. This is the closest related pattern. At process creation time rights are assigned to the process.
- Controlled Virtual Address Space [Fer02]. A VAS is assigned to each process that can be accessed according to the rights of the process.
- Authorization [Sch06]. Defines the rights to access resources.
- The Reference Monitor pattern, used to enforced the defined rights [Sch06].

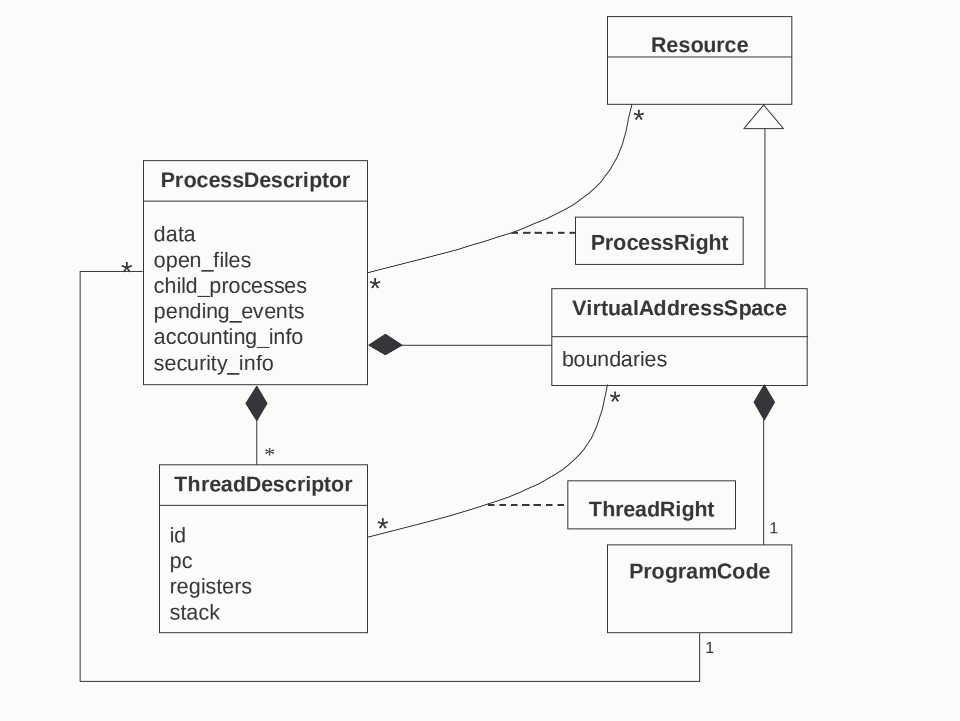


Figure 4. Class diagram for Secure Thread

3 Virtual address space structure

Virtual memory allows the total size of the memory used by processes to exceed the size of physical memory. Upon use, the virtual address is translated by the Memory Management Unit (MMU) to obtain a physical address that is used to access physical memory. As indicated earlier, to execute a process, the kernel creates a per-process virtual address space. The organization of each process' virtual address space (VAS) has an effect on performance and security. The organization of the VAS is defined by the hardware architecture. How do we select an appropriate virtual space structure to obtain specific security features?

3.1 Example

We have a system running applications using images requiring large graphic files. The application also has stringent security requirements. We need to decide on an appropriate VAS structure

3.2 Context

Multiprogramming systems with a variety of users and applications. Each process runs in its own address space. Processes execute on behalf of users and at times must be able to share memory areas, other times must be isolated, and in all cases we need access control. Performance is also an issue.

3.3 Problem

We need to select the virtual address space for processes depending on the majority of applications. Otherwise, we can have mismatches that may result in poor security or performance.

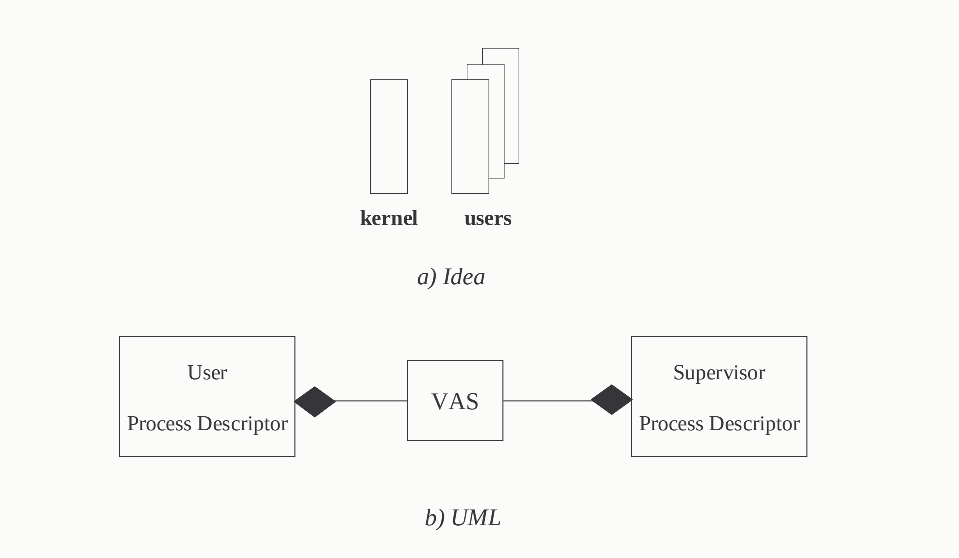
The possible solution is constrained by the following forces:

- Each process needs to be assigned a relatively large VAS to hold its data, stack, space for temporary variables, variables to keep the status of its devices, and other information.
- In multiprogramming environments processes have diverse requirements; some require isolation, others information sharing, others good performance.
- Data typing is useful to prevent errors and improve security. Several attacks occur by executing data and modifying code [Gol06].
- Sharing between address spaces should be convenient. Otherwise performance may suffer.

3.4 Solution

Select from four basic approaches that differ in their security features:

- *One address space per process* (Figure 5). The supervisor and each user process get their own address spaces.



a) Idea

b) UML

Figure 5. One address space per process

- *Two address spaces per process* (Figure 6). Each process gets a data and a code virtual address space.

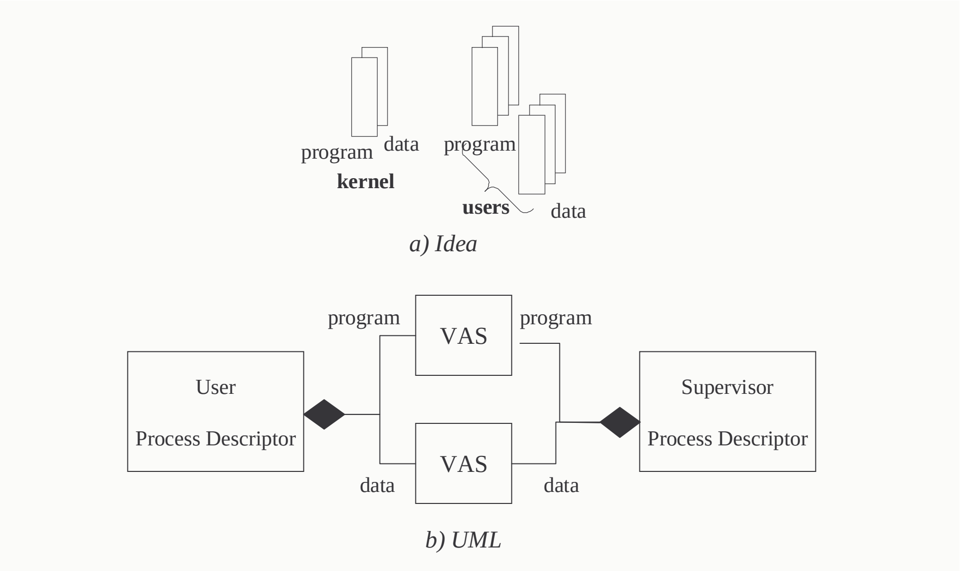


Figure 6. Two address spaces per process

- *One address space per user process, all of them shared with one address space for the OS* (Figure 7). The OS (supervisor) can be shared between all processes.

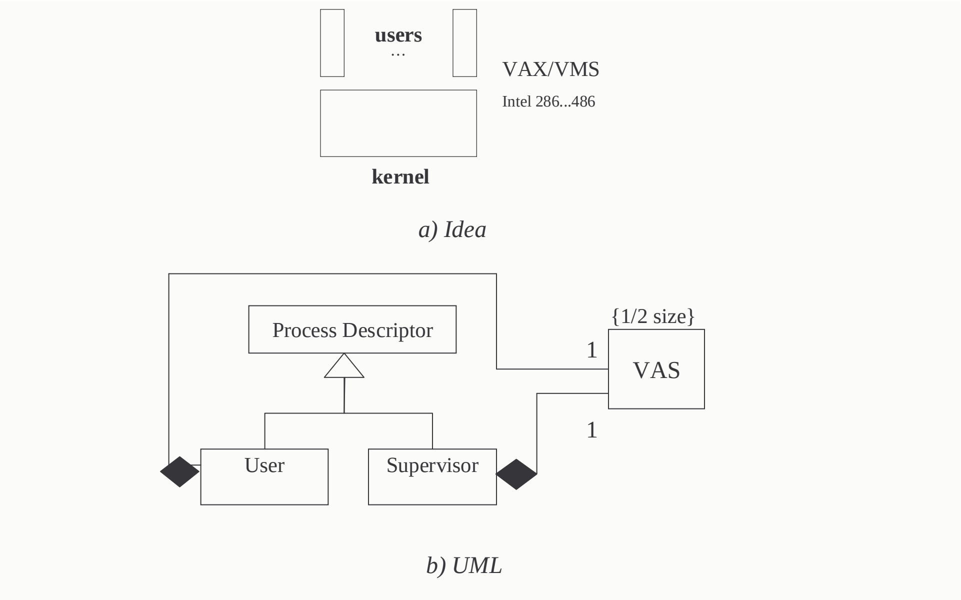


Figure 7. One address space per user process, all of them shared with one address space for the OS

- *A single-level address space* (Figure 8). Everything, including files, is mapped to this memory space.

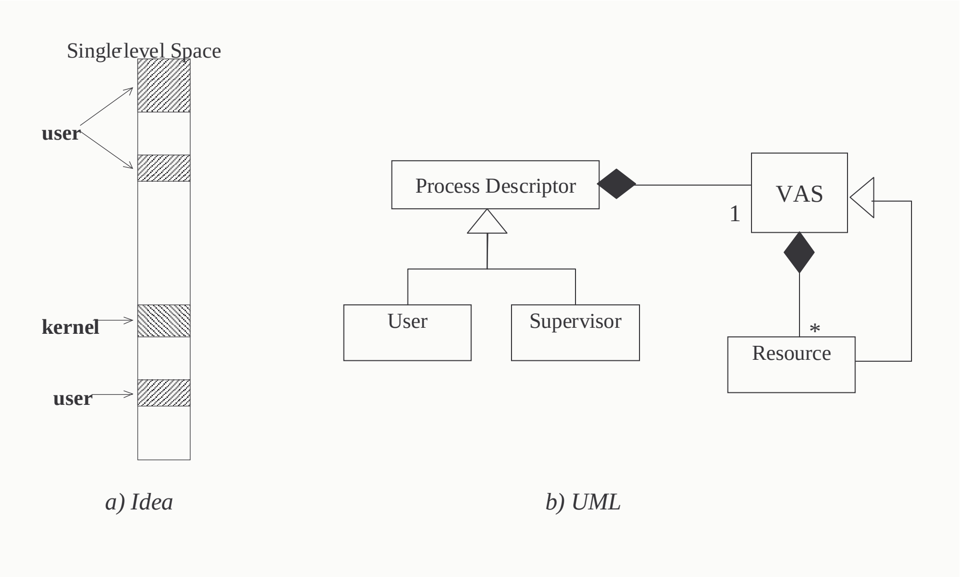


Figure 8. A single-level address space

Use of one VAS per process has the following tardeoffs:

- Good process isolation
- Some protection against the OS
- Simplicity
- Sharing is complex (special instructions to cross spaces are needed).

Use of two VAS per process has the following tradeoffs:

- Good process isolation
- Some protection against the OS
- Data and instructions can be separated for better protection (some attacks take advantage of execution of data or modification of code). Data typing is also good for reliability.
- A disadvantage is complex sharing plus rather poor address space utilization.

Use of one address space per user process, all of them shared with one address space for the OS has the following tradeoffs:

- Good process isolation
- Good sharing of resources and services (browsers, media players).
- This is not the best with respect to security (the supervisor has complete access to the user processes and it must be trusted).
- Another disadvantage is that the address space available to each user process has now been halved

Use of a single-level address space has the following tradeoffs:

- Good process isolation
- Logical simplicity
- Uniform protection (all I/O is mapped to memory)
- This is the most elegant solution (only one mechanism to protect memory and files), and potentially the most secure if capabilities are also used.

- It is hard to implement in hardware due to the large address space required.

3.5 Implementation

The VAS is implemented by the hardware architecture. The OS designer can choose one of the architectures based on the requirements of the applications according to the tradeoffs discussed above. In a particular case, the choice may be influenced by company policies, cost, performance, and other factors as well as security.

3.6 Known uses

- *One address space per process*. The NS32000, WE32100, and Clipper microprocessors.
- *Two address spaces per process*. This is used in the Motorola 68000 series.
- *One address space per user process, all of them shared with one address space for the OS.* This is used in the VAX series and in the Intel processors.
- *A single-level address space.* Multics, IBM S/38, IBM S/6000, and HP's PA-RISC use this approach.

3.7 Consequences

In addition to the specific consequences described as part of the solution we have the following general consequence:

- Without hardware support it is not feasible to separate the virtual address spaces of the processes. Most processors use register pairs or descriptors that indicate the base (start) of a memory unit and its length or limit. [Sil05].

3.8 Related patterns

- Secure Process
- Controlled Virtual Address Space [Fer02, Sch06]. A VAS is assigned to each process that can be accessed according to the rights of the process. The Virtual Address Space Structure complements that pattern.

4. Administrator Hierarchy

How do we limit the access rights of administrators? Define a hierarchy of system administrators with rights controlled using a Role-Based Access Control (RBAC) model.

4.1 Example

Unix defines a superuser who has all possible rights. This is convenient, for example, when somebody forgets a password, but allows hackers to totally control the system through a variety of implementation flaws. Through gaining access to the Administrator interface by logging through an account assigned the Administrator role, an individual can create new Administrator and User accounts, restrict their privileges and quotas, access their protected areas, and remove their accounts.

4.2 Context

An operating system with a variety of users, connected to the Internet. There are commands and data that are used for system administration and access to their use needs to be protected. This control is usually applied through special interfaces. There are at least two roles required to properly manage privileges, *Administrator* and *User*.

4.3 Problem

Usually, the administrator has rights such as creating accounts and passwords, installing programs, etc. This brings upon a series of security problems. A rogue administrator can do all the usual functions and even erase the log to hide his tracks. A hacker that takes over administrative power can do similar things. How do we curtail the excessive power of administrators to control rogue administrators or hackers?

The possible solution is constrained by the following forces:

- Administrators need to use commands that permit management of the system, e.g., define passwords for files, define quotas for files, and create user accounts. We cannot eliminate these functions.
- Administrators need to be able to delegate domain of responsibilities and privileges to manage those domains. They also need the right to take back these delegations. Otherwise, the system is too rigid.
- Administrators should have no control of system logs or no valid auditing would be possible.
- Administrators should have no access to the operational data in the users' applications.

4.4 Solution

Use the principle of separation of duty, where a user cannot perform critical functions unless in conjunction with others. Separate the different administrative rights into several hierarchical roles. The rights for these roles allow the administrators to perform their administrative functions and no more. Critical functions may require more than one administrative role to participate.

Structure

Figure 9 shows a hierarchy for administration roles. This follows the Composite pattern [Gam95], i.e., a role can be simple or composed of other roles, defining a tree hierarchy. The top-level administrator can add or remove administrators of any type and initialize the system but should have no other functions. After that, administrators in the second level control different aspects, e.g. security or use of resources. Administrators can further delegate their functions to lower-level administrators. Some functions may require two administrators to collaborate.

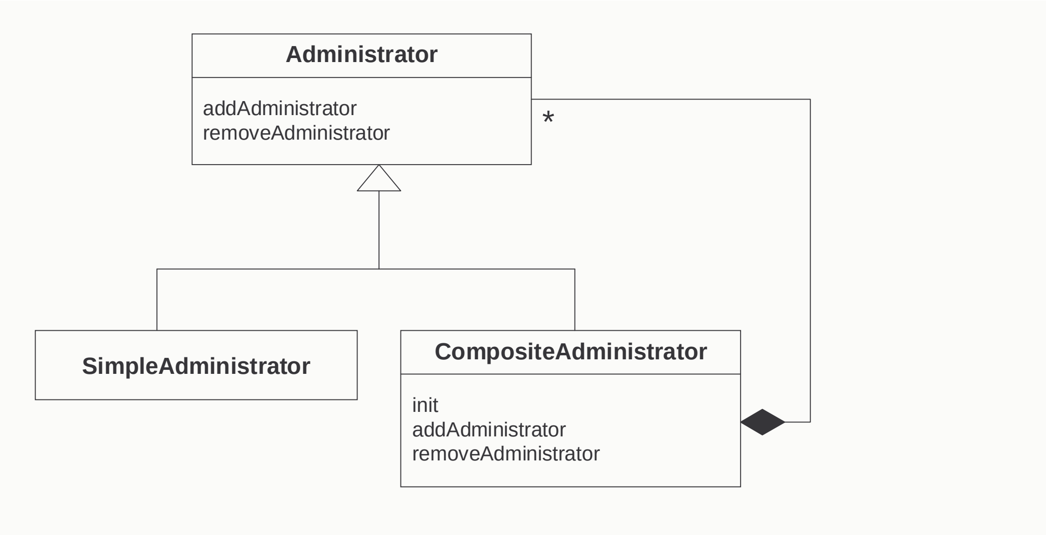


Figure 9. Class diagram for Administrator Structure

4.5 Implementation

Define a top administrative role to with the only function of setting up and initializing the system. This includes definition of roles, assignment of rights to roles, and assignment of users to roles. Separate the main administrative functions of the system and define an administrative role for each one of them. These define the second level of the hierarchy. Define further levels to accommodate administrative units in large systems. Figure 10 shows a typical hierarchy. Here the system administrator starts the system and does not perform later actions, the second-level administrator can perform set up and other functions, the security administrator defines security rights. Security Domain administrators define security in their domains. Other examples are shown in Section 4.7.

4.6 Example resolved

Some secure Unix versions such as Trusted Solaris use this approach. Now the superuser only starts the system. During execution time the administrators have restricted powers. If a hacker takes over his functions he can do only limited damage.

4.7 Known uses

- AIX [Cam90] reduces the privileges of the system administrator by defining five partially-ordered roles: Superuser, Security Administrator, Auditor, Resource Administrator, and Operator.
- Windows Windows NT uses four roles for administrative privileges: standard, administrator, guest, and operator. A User Manager has procedures for managing user accounts, groups, and authorization rules.
- Trusted Solaris [Sun04] This is an extension of Solaris 8. They use the concept of Trusted Roles with limited powers.
- *Argus Pitbull* [Arg] Least privilege applied to all processes, including the superuser. The superuser is implemented using three roles: Systems Security Officer, System Administrator, and System Operator.

4.8 Consequences

The Administrator Hierarchy pattern has the following advantages:

- If an administrative role is compromised, the attacker gets only limited privileges. The potential damage is limited.
- The reduced rights also reduce the possibility of misuse by the administrators.
- The hierarchical structure allows taking back control of a compromised administrative function.
- The advantages of the RBAC model apply: simpler and fewer authorization rules, flexibility for changes, etc. [Sch06].

Possible disadvantages include:

- Extra complexity for the administrative structure.
- Less expediency. Performing some functions may involve more than one administrator.
- Many attacks are still possible; if someone misuses an administrative right this pattern only limits the damage. Logging can help misuse detection.

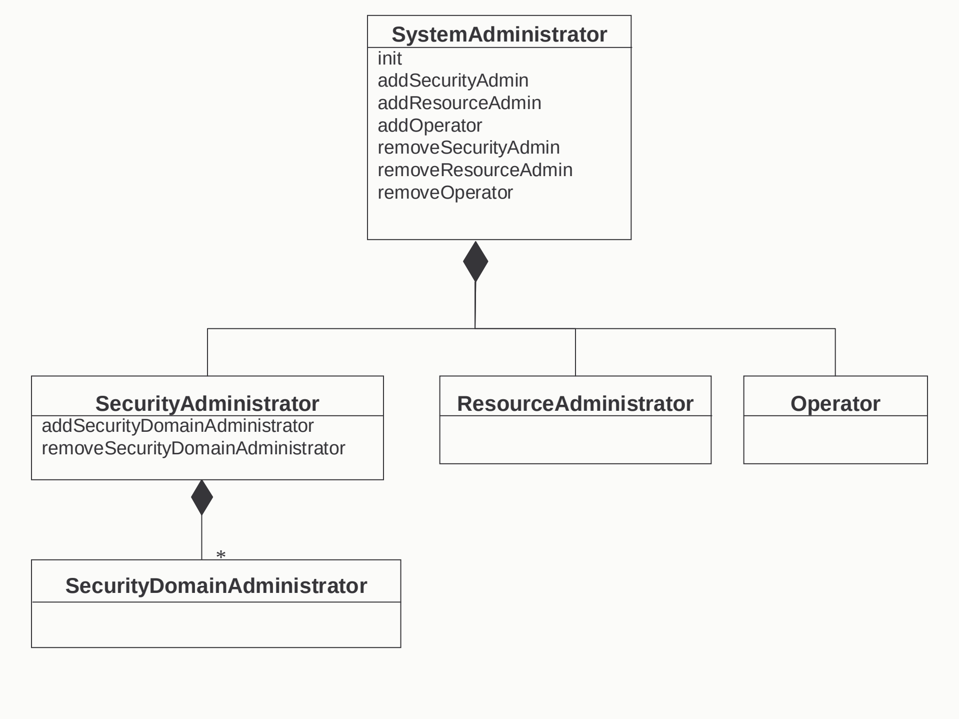


Figure 10. A typical administration hierarchy.

4.9 Related Patterns

This pattern applies the principles of least privilege and separation of duty (some people consider them patterns also). Each administrator role is given only the rights it needs to perform its duties and some functions may require collaboration.

Administrative rights are usually organized according to a RBAC model, a pattern for this model is given in [Fer01, Sch06].

Acknowledgements
Our shepherd, Juha Parssinen, provided valuable suggestions that considerably improved this paper. Paris Avgeriou gave valuable comments to an early version of this pattern.

References

[Arg] Argus Systems Group, “Trusted OS security: Principles and practice”, http://www.argus-systems.com/products/white_paper/pitbull

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *Pattern-Oriented Software Architecture: A System of Patterns*, Volume 1. J. Wiley, 1996.

[Cam90] N.A.Camillone , D.H.Steves, and K.C.Witte, “AIX operating system: A trustworthy computing system”, in *IBM RISC System/6000 Technology*, SA23-2619, IBM Corp., 1990, 168-172.

[Fer02] E.B.Fernandez, "Patterns for operating systems access control", *Procs. of PLoP 2002*, http://jerry.cs.uiuc.edu/~plop/plop2002/proceedings.html

[Fer03] E. B. Fernandez and J. C. Sinibaldi, “More patterns for operating system access control”, *Proc. of the 8th European conference on Pattern Languages of Programs, EuroPLoP 2003*, http://hillside.net/europlop, 381-398.

[Fer05] E.B.Fernandez and T. Sorgente, "A pattern language for secure operating system architectures” , *Procs. of the 5th Latin American Conference on Pattern Languages of Programs*, Campos do Jordao, Brazil, August 16-19, 2005.

[Fer06a] E.B.Fernandez, “Operating system access control”, Chapter 10 in [Sch06].

[Fer06b] E.B.Fernandez, E. Gudes, and M. Olivier, *The design of secure systems*, to be published by Addison-Wesley.

[Fri98] A. Frisch, *Essential Windows NT System Administration*, O’Reilly and Associates, Inc., Sebastopol, California, 1998.

[Gam95] E. Gamma, R. Helm,R. Johnson, and J. Vlissides, *Design patterns –Elements of reusable object-oriented software*, Addison-Wesley 1995.

[Gol06] D. Gollmann, *Computer security*, 2nd Ed., Wiley, 2006.

[Nut03] G. Nutt, *Operating systems* (3rd Ed.), Addison-Wesley, 2003.

[Nyh05] L. Nyhoff, *C++: An introduction to data structures (2nd Ed.)*, Prentice-Hall 2005.

[Pfl03] C.P.Pfleeger, *Security in computing*, 3rd Ed., Prentice-Hall, 2003. http://www.prenhall.com

[Sch00] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-oriented software architecture, vol. 2* , *Patterns for concurrent and networked objects*, J. Wiley & Sons, 2000.

[Sch06] M. Schumacher, E.B. Fernandez, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating security and systems engineering*, J. Wiley & Sons, 2006.

[Sil05] A. Silberschatz, P. Galvin, G. Gagne, *Operating System Concepts (7th Ed.)*, John Wiley & Sons, 2005

[Sun04] Trusted Solaris Operating System, http://www.sun.com/software/solaris/trustedsolaris/

[Sym01] http://www.symbian.com/developer/