Even more patterns for secure operating systems

Eduardo B. Fernandez
Florida Atlantic University
Dept. of Computer Science & Eng.
PO Box 9091, Boca Raton, FL 33431
+1 561 297-3466
ed@cse.fau.edu

Tami Sorgente
Florida Atlantic University
Dept. of Computer Science & Eng.
PO Box 9091, Boca Raton, FL 33431
+1 561 297-1392
tami@cse.fau.edu

Maria M. Larrondo-Petrie
Florida Atlantic University
Dept. of Computer Science & Eng.
PO Box 9091, Boca Raton, FL 33431
+1 561 297-3899
petrie@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 to Internet-based systems 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 for the representation of processes and threads, emphasizing their security aspects. Another pattern considers the selection of virtual address space structure. We finally present a pattern to control the power of administrators, a common source of security problems.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures – *patterns*; D4.6 [**Operating Systems**]: Security and Protection – *access control, authentication, information flow controls.*

General Terms
Documentation, Design, Security.

Keywords
Software Architectures, Patterns, Security, Operating Systems.

1. INTRODUCTION
The operating system (OS) acts as an intermediary between the user of a computer and the hardware. Its main purpose is to provide an environment in which users can execute programs in convenient and efficient manner, i.e. a high-level abstract machine. OSs also control and coordinate the available resources. 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.

Proceedings of the 13th Conference on Pattern Languages of Programs (PLoP2006), October 21-23, 2006, Portland, Oregon, USA

We have presented several patterns for different aspects of the security of operating systems [1, 2, 3, 4, 5]. These are patterns intended for designers of such systems. OS designers are usually experts on systems programming but know little about security, the use of patterns may help them build secure systems. These patterns are also useful for teaching security, we use them in our security courses and in a coming textbook [6]. Our previous patterns covered a range of security problems but there are still some aspects that we did not consider and we present here security patterns for three additional aspects. We assume the reader to be familiar with operating systems and with basic security concepts [6, 7, 8]. Figure 1 shows the relationships of the new patterns with respect to each other and with respect to some of our previous patterns (the patterns presented here are shown with double lines). Their thumbnail descriptions are given below, starting with the three new patterns:

Secure Process /Thread. How do we make sure that a process does not interfere with other processes or misuse shared resources? A process is a program in execution, a secure process is also a unit of execution isolation as well as a holder of rights to access resources. A secure process has a separate virtual address space and a set of rights to access resources. A thread is a lightweight process. A variant, called secure thread is a thread with controlled access to resources.

Virtual Address Space Selection. 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. The organization of each process' virtual address space (VAS) is defined by the hardware architecture and has an effect on performance and security. Consider all the hardware possibilities and select according to need.

Administrator Hierarchy. Many attacks come from the unlimited power of administrators. How do we limit this power? Define a hierarchy of system administrators with rights controlled using a Role-Based Access Control (RBAC) model and assign rights according to their functions.

Controlled Virtual Address Space [1]. 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 [2]. How to define the rights to be given to a new process? Define rights as part of its creation and give it a predefined subset of its parent's rights.

Authorization [5]. 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.

Role-Based Access Control [5]. 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 [5]. 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 authorization rules.

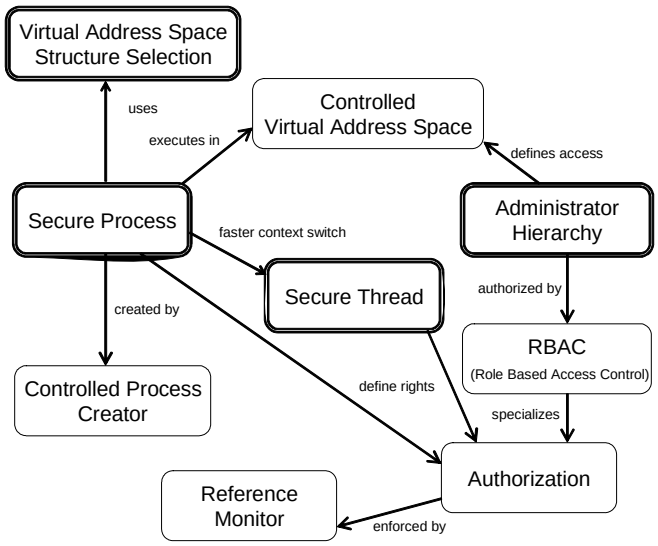


Figure 1. Pattern diagram for the patterns discussed here (double lined) and their relationship to already existing patterns

Going back to Figure 1, the central pattern in this paper is the Secure Process, which defines the conditions for processes to be secure. A variant of this pattern is the Secure Thread. An important aspect of the security of processes is their address space structure and one of these patterns (Virtual Address Space Selection) indicates criteria to select an appropriate virtual address space based on the expected types of applications. Whatever address space is chosen, this must be controlled and the Controlled Virtual Address Space pattern indicates its features. Another important pattern to have secure processes is the Controlled Process Creator that defines how the rights of a process are initially acquired. The Administrator Hierarchy pattern restricts the rights of administrators to prevent some attacks. This pattern defines rights for administrators, including their rights to access virtual address spaces (as well as other resources). The rights for processes or threads are defined by the Authorization pattern and enforced by a Reference Monitor pattern. One way to organize authorization rights, typically used for administrators is Role-Based Access Control (RBAC).

Section 2 presents the Secure Process pattern and its variant the Secure Thread. Virtual Address Space Structure Selection is described in Section 3, while Administrator Hierarchy is presented in Section 4. We end with some conclusions.

2. SECURE PROCESS

How do we make sure that a process does not interfere with other processes or misuse shared resources? A process is a program in execution. A secure process is given its own virtual address space and a set of rights to access resources.

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). While performance was good, once its poor security was known nobody wanted to use this operating system.

Context

Typically, operating systems support a multiprogramming environment with several user-defined and system processes and threads active at a given time. A process is a program in execution and a thread is a lightweight process. During execution it is essential to maintain all information regarding the 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.

Problem

We need to control the resources accessed by a process during its execution and protect its context from other processes. The resources that can be accessed by a process define its *execution domain* and the process should not break the boundaries of this domain. The integrity of a process' context is essential not only for context switching, but also for security (so it cannot be disrupted by another process) and for reliability to prevent a rogue process from interfering with other processes.

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

- Each process requires some data, a stack, space for temporary variables, keeping the status of its devices and other information. All this information resides in its address space and needs to be protected.
- If processes have unrestricted access to resources they can interfere with the execution of other processes and misuse shared resources. 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 or least privilege principle [7]).
- The rights assigned to a process should be fine-grained. Otherwise we cannot apply the least privilege principle.

Solution

Assign to each process a set of authorization rights to access the resources they need. Assign also to the process a unique address space to store its context as well as its needed execution-time data. This protects processes from interference from the other processes, assuring confidentiality and integrity of the shared data and proper use of shared resources. In the **ProcessDescriptor,** a data structure containing all the information a process needs for its execution, add rights to make access to any resource explicitly authorized. Every access to a resource is intercepted and checked for authorization. It may also be possible to add resource quotas to avoid denial of service problems but this requires some global resource usage policies.

Structure

In Figure 2 each **ProcessDescriptor** has **ProcessRights** for specific **Resources.** Additional security information indicates the **Owner** of the process (when a process is created it receives rights from its owner or its father process). The process rights are defined by the Authorization pattern (the Process Descriptor acts as subject in this pattern) and are enforced by the Reference Monitor pattern, which intercepts request for resources and checks them for authorization. More than one ProcessDescriptor can be created, corresponding to multiple executions of **ProgramCode,** and describing different processes. A separate **VirtualAddressSpace** is associated with each process (defined by the Controlled Virtual Address Space pattern). The process context is stored in the VirtualAddressSpace of the process, while the ProgramCode can be shared by several processes.

Dynamics

Figure 3 shows a sequence diagram for the use case "Access a resource". A **requestResource** operation from a process includes the process ID and the intended type of access. The request is intercepted by the Reference Monitor which determines if it is authorized (**checkAccess** operation in the ProcessRight). If it is, the access proceeds.

Other related use cases (not shown) include "Assign a right to a process" and "Remove a right from a process".

Implementation

The Process Descriptor is typically called Process Control Block (PCB), or Task Control Block (TCB), and it is realized as a data structure that includes references (pointers) to its code section, its stack, and other needed information. There are different alternatives to implement data structures in general [9]. Records (`structs` in C) are typically used for the Process Descriptor. The Process Descriptors of the processes in the same state, e.g. ready or waiting, 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 a virtual address space to a process (see the pattern Virtual Address Space Selection below). There are also different ways to associate rights with a new process (see the Controlled

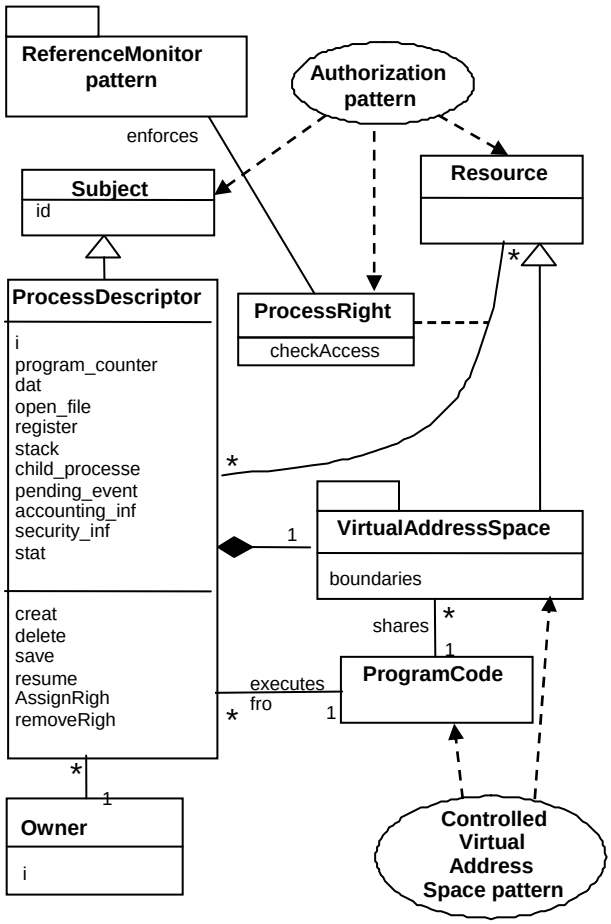


Figure 2. Class diagram for Secure Process

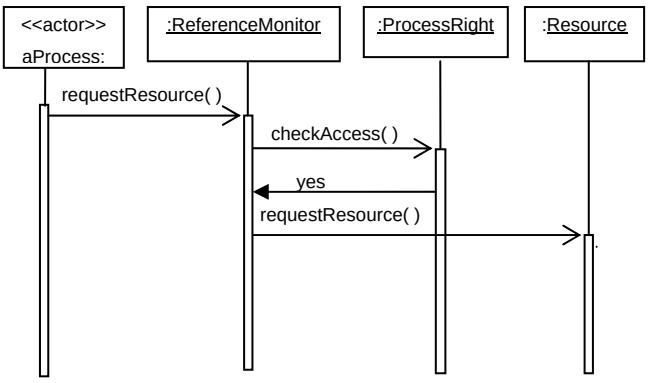


Figure 3. Sequence diagram for use case "Access a resource".

Process Creator in [2]). The hardware architecture normally implements the virtual address space and restricts access to the sections (segments) allocated to each process using appropriate mechanisms.

The pattern models as shown represent models where subjects have rights described by an Access Matrix or according to Role-Based Access Control [10, 5]. Some operating systems use Multilevel (typically mandatory) models where the access of a process is decided by its level with respect to the level of the accessed resource. In multilevel systems, 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.

Example resolved

Company X solved its problem by adding rights to a process representation. Now each process is constrained to access only those resources for which it has rights. This protects each process from each other as well as the confidentiality and integrity of shared data and other resources. While other security problems may still persist, the general security of the OS increased significantly

Variant

Secure Thread. Because of the slow context switching of processes, most operating systems use threads, which have a smaller context and share the same VAS. A secure thread is a thread with controlled access to 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 and shared by peer threads. **ThreadRights** define access rights to the VAS.

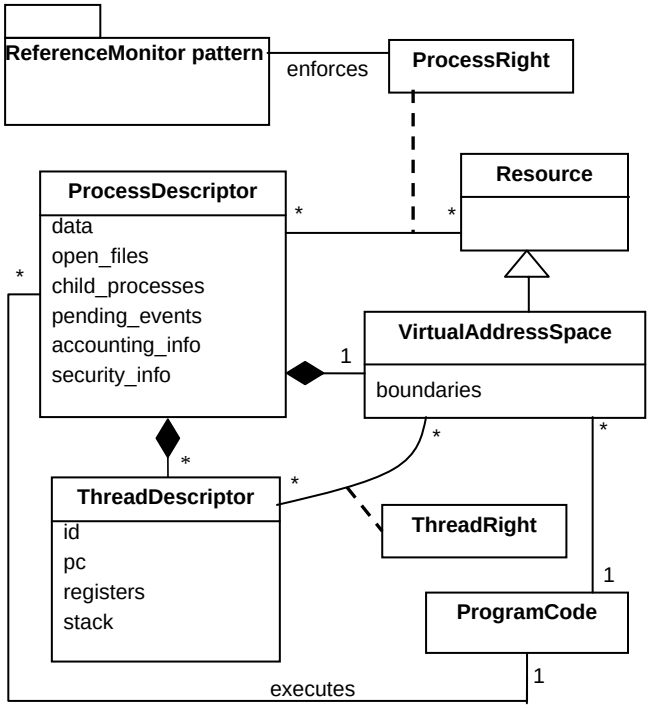


Figure 4. Class diagram for Secure Thread

Thread status includes typically a stack, a program counter, and some status bits. There are different ways to associate threads with a process [11]. Typically, several threads are collected into a process. Threads can be created with special packages, e.g., PTHREADS 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).

Known uses

- Linux uses records for process descriptors. One of the entries defines the process credentials (rights) that define its access to resources [12, 11]. Other entries describe its owner (subject) and process id. A more elaborate approach, with execution domains, is used in Selinux, a secure version of Linux 13.
- Windows NT and 2000 define resources as objects (really as classes). The process id is used to decide access to objects [11]. Each file object has a security descriptor that 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 standard [11].
- 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).

Consequences

This pattern has the following advantages:

- It is possible to give precise specific rights (access types) for resources to each process, which restricts them to access only authorized resources.
- It is possible to apply the least privilege principle by appropriate distribution of access rights for execution.
- The process' contexts can be protected from other processes, because they are restricted to access only authorized resources.
- The virtual address space of a process can be protected by the hardware and its memory manager.
- It is possible to stop or mitigate attacks coming from rogue processes (produced by malicious users or malware).

This pattern has the following disadvantages:

- There is some overhead in using a Reference Monitor to enforce accesses.
- It may not be clear what specific rights to assign to each process.
- Having a separate address space implies a slow context switch, which affects performance. Because of this, kernel processes are usually implemented as threads and share an address space, which reduces their security.
- There are other security problems not controlled this way, e.g., denial of service, users taking control in administrator mode, virus propagation. Those problems require complementary security mechanisms, some of which are described by other patterns [5].

Related patterns

- Controlled Process Creator [2, 5]. At process creation time, rights are assigned to the process.

- Controlled Virtual Address Space [1, 5]. A VAS is assigned to each process that can be accessed according to the rights of the process.
- Authorization [10, 5]. Defines the rights to access resources.
- The Reference Monitor pattern, used to enforced the defined rights [5].

3. Virtual Address Space Selection

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. The organization of each process' virtual address space (VAS) is defined by the hardware architecture and has an effect on performance and security. Consider all the possibilities, displayed side to side, and select according to need.

Example

We have a system running applications using images requiring large graphic files. The application also has stringent security requirements because some of the images are sensitive and should be only accessed by authorized users. We need to decide on an appropriate VAS structure

Context

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 Address Translation Unit (usually called Memory Management Unit (MMU) in microprocessors) 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. We need to accommodate a multiprogramming system with a variety of users and applications. 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 may also be an issue.

Problem

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

The possible solution is constrained by the following forces:

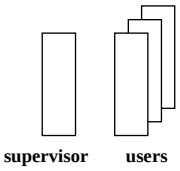
- Each process needs to be assigned a relatively large VAS to hold its data, stack, space for temporary variables, variables to keep the status of its devices, and other information.
- In multiprogramming environments processes have diverse requirements; some require isolation, others information sharing, others good performance.
- Data typing is useful to prevent errors and improve security. Several attacks occur by executing data and modifying code [7].
- Because of the need to share the kernel services and its utilities (databases, editors, etc), sharing between address spaces should be fast and convenient. Otherwise performance may suffer.
- In order to decide, one should have a good knowledge of the type of applications to be executed.

Solution

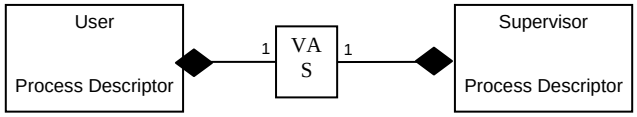
Select from four basic approaches that differ in their security features:

One address space per process (Figure 5). The supervisor (kernel plus utilities) and each user process get their own address spaces. Use of one VAS per process has the following tradeoffs:

- Good process isolation. Each process context is in a separated VAS.
- Some protection against the OS. There is a well-defined interface between the process and the OS where checks can be applied.
- Simplicity in allocating VASs.
- Sharing is complex (special instructions to cross spaces are needed). All processes use the same addresses and interprocess communication requires specification of the intended VAS. We need special instructions to move across VASs (overhead).
- Other resources may need special protection mechanisms (they may not be mapped to memory).



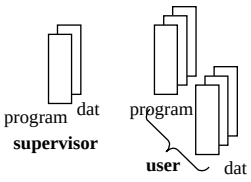
a) Idea



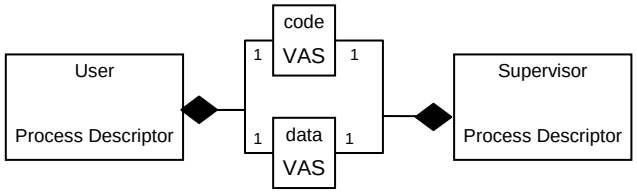
b) UML

Figure 5. One address space per process

Two address spaces per process (Figure 6). Each process gets a data and a code (program) virtual address space. Use of two VASs per process has the following tradeoffs:



a) Idea



b) UML

Figure 6. Two address spaces per process

- Good process isolation. Each process context is in a separated VAS.
- Some protection against the OS. There is a well-defined interface between the process and the OS where checks can be applied.
- 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.
- Complex sharing plus rather poor address space utilization. We need special instructions to move across VASs (overhead). If there is little code or little data we cannot allocate them in the same address space.
- Other resources may need special protection mechanisms (they may not be mapped to memory).

One address space per user process, all of them shared with the address space of the OS (Figure 7). The OS (supervisor) can be shared between all processes. 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, but only between user processes.
- Good sharing of resources and services (browsers, media players). Supervisor is in the same VAS.
- The supervisor has direct access to the user processes and it can misuse their information or interfere with their execution.
- The address space available to each user process has now been halved.
- Other resources may need special protection mechanisms (they may not be mapped to memory).

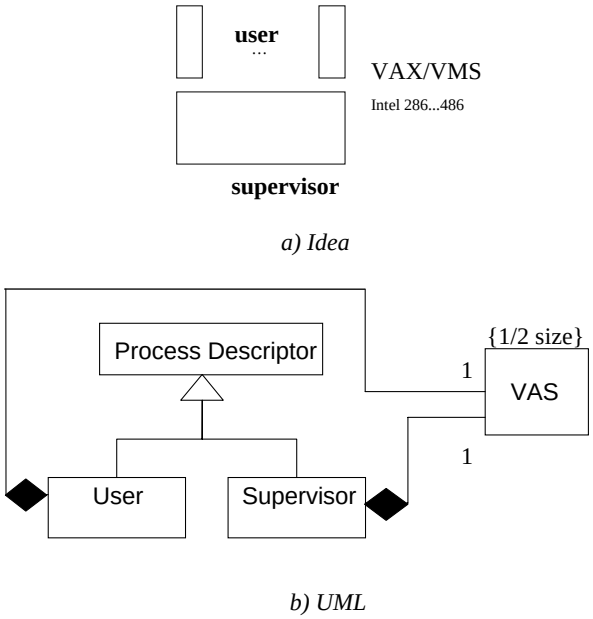


Figure 7. One address space per user process. All of them shared with the address space of the operating system.

A single-level address space (Figure 8). Everything, including files and I/O, is mapped to this memory space. Use of a single-level address space has the following tradeoffs:

- Good process isolation. Descriptors are a good mechanism for separating VAS areas.
- Logical simplicity. Every resource is mapped to one VAS.
- Uniform protection. Every resource is protected in the same way.
- 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. The size of memory incurs on some extra performance overhead.

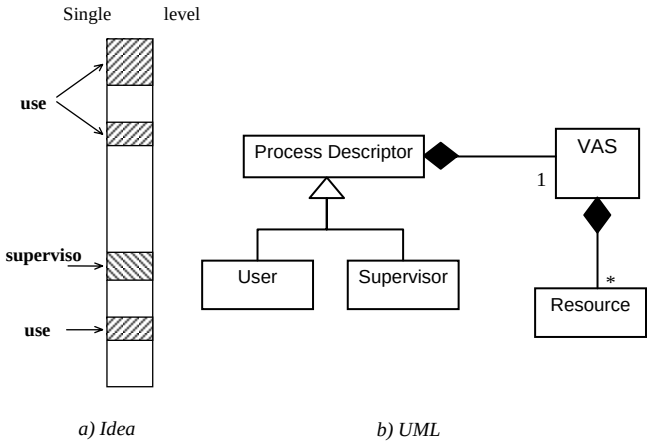


Figure 8. A single-level address space

Implementation

Most processors use register pairs or descriptors that indicate the base (start) of a memory unit (segment) and its length or limit [11]. VASs are implemented using indexes or hashing [11]. Both approaches imply some loss of performance in order to have a large address space. The OS designer can choose one of these architectures based on the requirements of the applications and according to the tradeoffs discussed earlier. A particular choice may be influenced by company policies, cost, performance, and other factors as well as security. The commercial availability of specific types of VASs is another issue, there are few hardware manufacturers and there may not be any processor with the required features.

Known uses

- *One address space per process*. This is used in the NS32000, WE32100, and Clipper microprocessors [14]. Several versions of Unix were implemented in these processors.
- *Two address spaces per process*. This is used in the Motorola 68000 series. The Minix 2 OS uses this approach [15].
- One address space per user process, all of them shared with one address space for the OS. This was used in the Digital Equipment's VAX series and is still used in the Intel processors. Typically, Windows run in this type of address space.
- *A single-level address space.* Multics, IBM's S/38, IBM's S/6000, and HP's PA-RISC use this approach [16]. Multics had

its own operating system. IBM's S/6000 run AIX, a version of Unix [17]. The PA-RISC architectures ran HP-UX, a version of Unix, but also Linux, OpenBSD, and Windows NT.

Consequences

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

- Without hardware support it is not feasible to separate the virtual address spaces of the processes. This means the OS designer has limited choices.
- If the mix of applications is not well defined, it is hard to select the best solution. Then, considerations other than security become more important.

Related patterns

- Secure Process. The interaction between processes depends strongly on the virtual address space configuration, which can affect security, performance, and sharing properties of the processes.
- Controlled Virtual Address Space [1, 5]. A VAS is assigned to each process that can be accessed according to the rights of the process. The Virtual Address Space Structure is applied first to select the appropriate structure. Once selected, the VAS is secured using the approach of the Controlled Virtual Space pattern.
- The Secure Layers pattern [18]. The VAS is assigned to one of the kernel layers but it needs to interact with other layers, including the hardware layer and the file layer.

4. Administrator Hierarchy

Many attacks come from the unlimited power of administrators. How do we limit the power of administrators? Define a hierarchy of system administrators with rights defined using a Role-Based Access Control (RBAC) model and assign rights according to their functions.

4.1 Example

Unix defines a superuser who has all possible rights. This is expedient; for example, when somebody forgets a password, but this approach allows hackers to totally control the system through a variety of implementation flaws. By gaining access to the Administrator rights, an individual can create new Administrator and User accounts, restrict their privileges and quotas, access their protected areas, and remove their accounts. A legitimate but bad-intentioned administrator can similarly do a lot of damage.

4.2 Context

An operating system with a variety of users, connected to the Internet. Special commands and data used for system administration need to be controlled. 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 creates 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 some responsibilities and privileges to manage large 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 because they could erase or modify these logs.
- Administrators should have no access to the operational data in the users' applications. If they do, their accesses should be logged.

4.4 Solution

Distribute the administrative rights into hierarchical roles. The rights for these roles allow the administrators to perform their administrative functions but no more. Critical functions may require more than one administrative role to agree. Use the principle of separation of duty [19], where a user cannot perform critical functions unless in conjunction with others. The hierarchical structure permits revocation of previously granted rights.

Structure

Figure 9 shows a hierarchy for administration roles. This follows the Composite pattern [20], 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. 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.

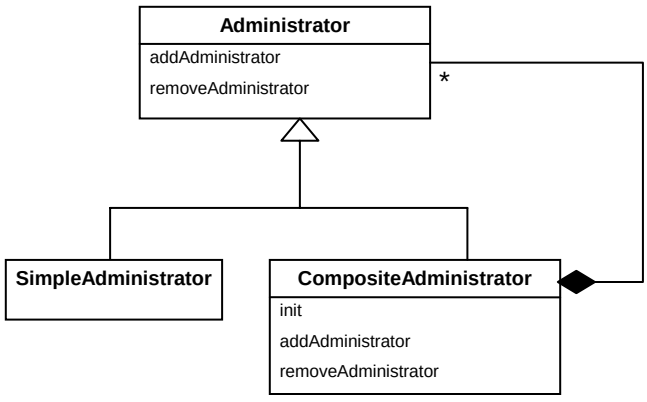


Figure 9. Class diagram for Administrator Structure

4.5 Implementation

Define a top administrative role with the only function of setting up and initializing the system. This includes definition of administrative 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 other levels to accommodate administrative units in large systems or for distributing rights into functional sets. 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.

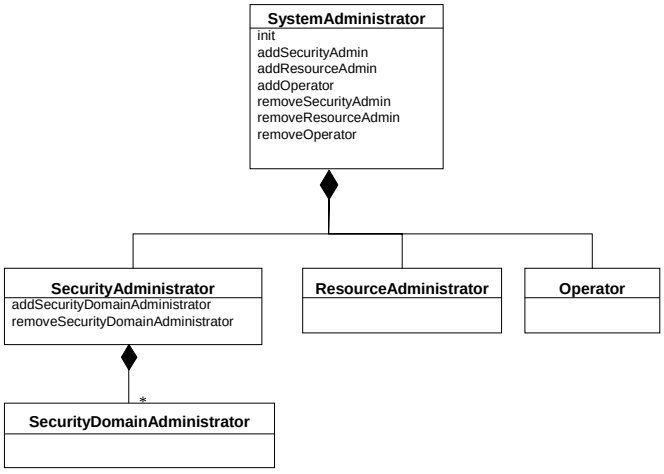


Figure 10. A typical administration hierarchy

4.6 Example resolved

Some secure Unix versions such as Trusted Solaris (see Section 4.7) use this approach. Now the superuser only starts the system. During normal operation the administrators have restricted powers. If a hacker takes over their functions he can do only limited damage.

4.7 Known uses

- AIX [17] reduces the privileges of the system administrator by defining five partially-ordered roles: Superuser, Security Administrator, Auditor, Resource Administrator, and Operator.
- 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 [21] is an extension of Solaris 8 operating system. It uses the concept of Trusted Roles with limited powers.
- Argus Pitbull [22] applies least privilege 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 legitimate 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 [5].
- This structure is useful not only for operating systems but also for servers, database systems, or any systems that require administration.

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 by keeping a record of all actions executed by administrators.
- Because some functions may require two administrators to agree to perform it, we need to add OCL constraints in the model to indicate this, increasing its complexity.
- In some cases, administrators need to communicate to perform their jobs but the pattern assumes no communication. This aspect can be corrected with a few associations.

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 [10, 5].

5. CONCLUSIONS

These patterns contribute three more solutions to help make operating systems more secure. The security of complex systems such as OSs is a difficult problem and more patterns are needed. A catalog of these patterns would be useful to operating system designers confronted with balancing the increasing functionality of these systems with the need to make them secure. Taken together, our four papers on operating system security patterns can form the basis of such a catalog. A related aspect is the security of the OS utilities and similar patterns may apply [23]. These patterns only apply to security aspects, clearly other aspects are important; the debate about the best OS architecture is not yet finished [24].

6. ACKNOWLEDGMENTS

Our shepherd Juha Parssinen, as well as Sami Lehtonen, Ralph Johnson, and Paris Avgeriou provided valuable suggestions that considerably improved this paper. Comments from the Secure Systems Research Group at Florida Atlantic University were also of great value. Finally, the PLoP 2006 Writer Workshop participants gave further useful advice.

This work was partially supported by funding from the Department of Defense.

7. REFERENCES

[1] Fernandez, E. B., 2002. Patterns for operating systems access control. In *Proceedings of 9th Conference on Pattern Language of Programs (PLoP 2002)* (Monticello, Illinois, USA, September 8-12, 2002). PLoP 2002. http://jerry.cs.uiuc.edu/~plop/plop2002/proceedings.html

[2] Fernandez, E. B. and Sinibaldi, J. C. 2003. More patterns for operating system access control. In *Proceedings of the 8th European conference on Pattern Languages of Programs*, (Irsee, Germany, June 25-29, 2003). EuroPLoP 2003, 381-398, http://hillside.net/europlop.

[3] Fernandez, E. B. and Sorgente, T. 2005. A pattern language for secure operating system architectures. In *Proceedings of the 5th Latin American Conference on Pattern Languages of Programs* (Campos do Jordao, Brazil, August 16-19, 2005). SugarloafPLoP 2005.

[4] Fernandez, E. B. 2006. Operating system access control. Chapter 10 in Schumacher, M. Fernandez, E.B. Hybertson, D. Buschmann, F. and Sommerlad, P. *Security Patterns: Integrating security and systems engineering*, J. Wiley & Sons, 2006.

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

[6] Fernandez, E. B., Gudes, E. and Olivier, M. 2009. *The Design of Secure Systems*, under contract with Addison-Wesley, to appear 2009.

[7] Gollmann, D. 2006. *Computer security*, 2nd Edition, Wiley, 2006.

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

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

[10] Fernandez, E. B. and Pan, R. Y. 2001. A pattern language for security models. In *Proceedings of 8th Conference on Pattern Language of Programs* (Monticello, Illinois, USA, September 11-15, 2001). PLoP 2001. http://hillside.net/plop/plop2001/

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

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

[13] Security Enhanced Linux, http://www.nsa.gov/selinux

[14] Fernandez, E.B. 1985. Microprocessor architecture: The 32-bit generation. In *VLSI Systems Design* (October 1985), 34-44.

[15] Tannenbaum, A.S., Herder, J.N. and Bos, H. 2006. Can we make operating systems reliable and secure? In *Computer* (May 2006), IEEE, 44-51.

[16] PA-RISC family, http://en.wikipedia.org/wiki/PA-RISC

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

[18] Yoder, J. and Barcalow, J. 2000. Architectural Patterns for Enabling Application Security. In *Proceedings of the 4tth Conference of Pattern Languages of Programs* (Monticello, Illinois, USA, September 3-5, 1997). PLoP 1997. Also Chapter 15 in *Pattern Languages of Program Design*, vol. 4 (N. Harrison, B. Foote, and H. Rohnert, Eds.), Addison-Wesley, 2000.

[19] Summers, R.C. 1995. *Secure computing: Threats and safeguards*, McGraw-Hill, 1997.

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

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

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

[23] Hafiz, M. and Johnson, R. 2008. Evolution of MTA Architecture: The Impact of Security. In *Software: Practice and Experience* (May 12, 2008), John Wiley & Sons, 2008. DOI= http://dx.doi.org/10.1002/spe.880

[24] Tanenbaum-Torvalds debate, http://en.wikipedia.org/wiki/Tanenbaum-Torvalds_debate