Chapter 4. Security in operating systems and hardware

The operating system (OS) is in direct contact with the hardware and supports all the services used by applications; its security is therefore fundamental. Its compromise can affect all applications and the persistent data. We consider first the needed hardware support and then we analyze how each operating system function provides security. We finally look at the security aspects of some commercial operating systems and describe some hardened versions of them. Since the operating system must work very closely with the hardware we also discuss hardware aspects in this chapter. At the end we consider the use of trusted hardware.

4.1 Overview

Operating systems act as an intermediary between the user of a computer and its hardware. The purpose of an operating system is to provide an environment in which users can execute programs in a convenient and efficient manner without being concerned with lower level details [Sil12]. Operating systems control and coordinate the available resources in order to present to the user an abstract machine with a uniform set of high-level features, independent of the hardware platform. The security of operating systems is very critical since the operating system supports the execution of all the applications and stores the persistent data. Most of the reported attacks to computer systems have occurred through the operating system.

Computer system functionality can be divided between the *kernel* (or operating system proper), and user-oriented *utilities* or services. Typically, an operating system includes the following functional components:

- *Process Management*-- handles creation and deletion of processes, communication and scheduling. Its security aspects include assignment of rights to new processes, secure communication between processes, and controlled execution environments for processes (See Sections 4.3 and 4.4).
- *Memory Management*-- keeps track of which parts of memory are used by which processes; that is, it allocates and deallocates memory. For security purposes, this function must provide isolation of process areas and controlled sharing of other areas (Sections 4.5 and 4.6).
- *File Management*-- handles creation and deletion of files and directories, file searches, and mapping files to secondary storage. The file manager must define and enforce authorizations or permissions to use the files (Section 4.8).
- *I/O Management*-- provides interfaces to hardware device drivers, as well as handling mass memory management components including buffering, caching, and spooling. I/O is usually mapped to memory and the memory manager controls its use.
- *Networking*-- controls communication paths between two or more systems. We discuss networking systems in Chapter 7.
- *User Interface*--communicates between user and OS including command interpreters. This function must authenticate users when they login to the system (Section 4.9).

Early attempts to build secure operating systems used the concept of *security kernel*, a combination of hardware and software that implemented a basic set of security requirements typically including a reference monitor and associated functions. As such, a security kernel must intercept all access requests, it must be protected from tampering, and be verifiably correct [Ame83, Fei79]. Typically, security kernels implement a Bell-LaPadula model, although this is not a requirement. A security kernel overlaps most of the functions of the operating system kernel but includes only security-related functions such as process creation, destruction, and domain switching, as well as memory protection (to protect the code and data of the kernel and the processes), and basic I/O functions that may access memory or files. The problem is that practical operating system kernels are too complex to be verified and it may be necessary to use entirely new systems to accommodate this approach. Current efforts, reviewed in Section 4.12, are based on the application of security principles to modify or extend commercial systems to improve their security.

A *Trusted Computing Base* (TCB) is the combination of all the security mechanisms that implement the security policies of a system, in this case the operating system. All the security functions we will identify in later sections are part of the TCB.

We start by defining requirements for the hardware to provide support for all these functions.

4.2 Requirements of hardware to support security

Being the lowest level of the system and the only one that actually performs physical actions, the hardware affects all computations. Because of this, it is important to have an appropriate platform if we want to have a secure system; it may not be possible to correct in the software what should have been done by the hardware. Even if we could, this would be at the cost of a high performance overhead. In general, the security of the operating system depends strongly on the support it gets from the hardware.

The tendency towards reduced Instruction Set Computers (RISC) was a blow against security, designers eliminated any features that they considered overhead, including most protection mechanisms. RISC machines rely heavily on compilers to optimize performance and there were thoughts about letting the compiler apply all security controls as well. However, a compiler cannot control security because it compiles each program in isolation and in addition, it cannot predict dynamic actions. Also, a typical system uses a variety of compilers, which cannot be guaranteed to be secure. This means that if we want security we need at least some basic functions to be provided by the hardware.

The hardware must at least support two basic operating system security policies:

- *Process isolation*—a process must be protected from interference from the other processes.
- *Controlled resource sharing*—processes must be able to share resources in a controlled way.

The operating system directs the hardware mechanisms in the application of these policies, using the hardware to restrict the use of instructions, to control the loading of process control registers, and to restrict access to areas of memory. From these two basic policies the operating system builds more elaborate policies, e.g., accountability and need-to-know.

In order to implement these basic policies, more specific requirements for secure hardware were defined in [Lan84] and extended here:

- *Define explicit processes and separate their domains*. A *domain* is a set of resources that can be used in specific ways by processes to perform their work, i.e. a set of rights that give processes access to some protection objects. There are four ways to perform separation of domains: physically, temporally, cryptographically, and logically. This separation is required to apply the need-to-know policy and for accountability purposes. We are concerned here mostly with logical separation of domains, where the separation is performed by assigning different sets of rights to different processes. We discuss this in detail in Sections 4.3 and 4.4.
- *Establish initial domain*. There must be an initial consistent state from which additional domains can be started and kept separated. Only special trusted programs should be able to invoke system initialization. Larger systems include a separate processor to perform initialization; these processors can be used to establish a secure initial domain. This is also called *safe restart* and is considered a basic principle for secure systems [Sha02]. Another aspect, very important to portable devices, is the assurance that a known hardware with a known operating system is running in an identified device [Har02] (See Section 4.)..
- *Link users with domains*. There must be an identifiable principal or responsible party; subjects associated with these principals perform resource requests. This is an application of the policy of accountability. Establishing these links is mostly done by software but the hardware provides a path between the user and the machine.
- *Control communication between domains*. Crossing from one domain to another should be strictly controlled through special instructions or gate crossings. This implies the concept of *protected entry points*, forcing the calls to processes in other domains to go to specific entry points. This is discussed further in Section 4.4.
- *Detect and handle faults*. Faults can be sources of security exposures. In particular, interrupt handlers should have their own execution domains.

4.3 Process and resource protection

The most basic function of an operating system is to create and manage *processes* (programs in execution). When a process is created it must be given access to resources in the form of rights to use these resources. The set of rights of the process define its execution domain. A process executes on behalf of a principal (user or role). The hardware must protect a process from other processes that intentionally or accidentally

may corrupt its execution context. This context includes code, program counter, stack, registers, status, and references to its resources (domain).

A process is defined by its *Process Control Block* (PCB), a data structure containing its id, and references to the parts of its context. A process should receive a separate address space for its execution. Some architectures, e.g., the Intel X86 series, have hardware support for the type of PCB used by common operating systems (this accelerates *context switching*, changing execution from one process to another at the expense of flexibility). A *thread* is a lightweight process (faster context switching than a process) and shares its address space with other threads. Typically, a thread includes a program counter, a register set, and a stack. Because of its shared address space, an error or attack from another thread can corrupt its memory. Thread stacks can be protected if they are kept in the system address space using separated segments or pages. Most modern operating systems, e.g., Solaris, allow several threads to be bundled in one process; this protects the thread group as a whole from other processes. User processes and threads can be created with special packages, e.g., Posix in Unix, or through the language, as in Java or Ada. The operating system defines kernel threads as units of concurrent execution. For the reasons discussed above, kernel threads usually don't have much protection against each other. Figure 4.1 shows the structure of a secure process [Fer13].

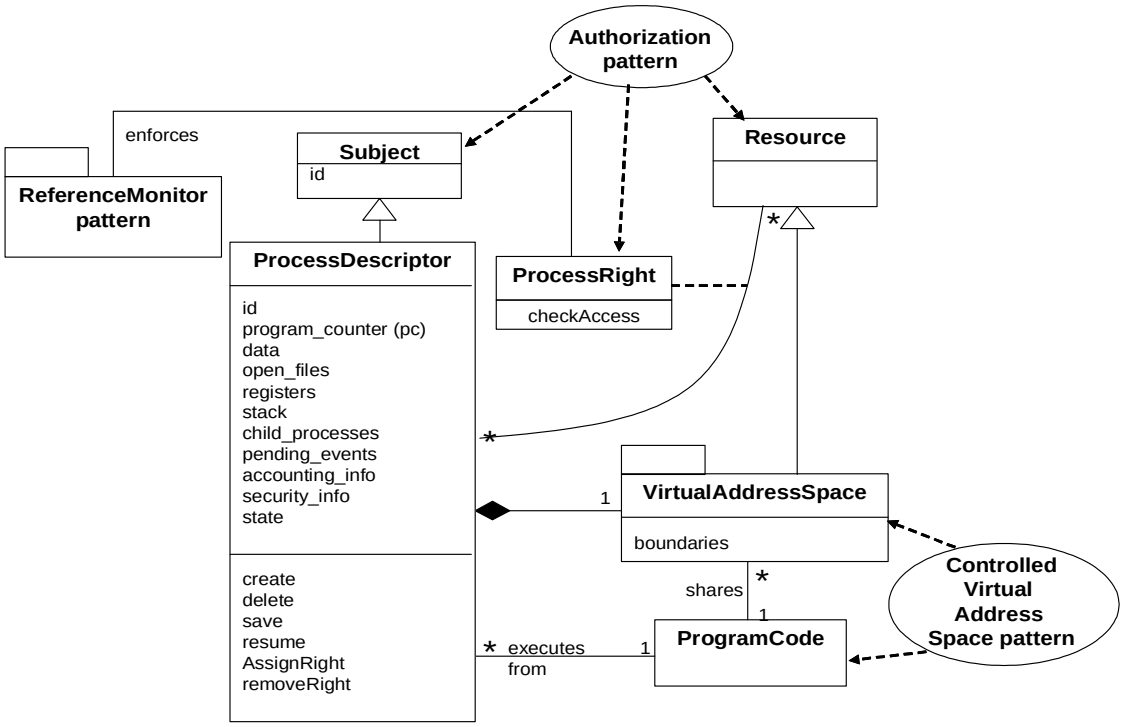

Fig. 4.1 Class diagram for Secure Process

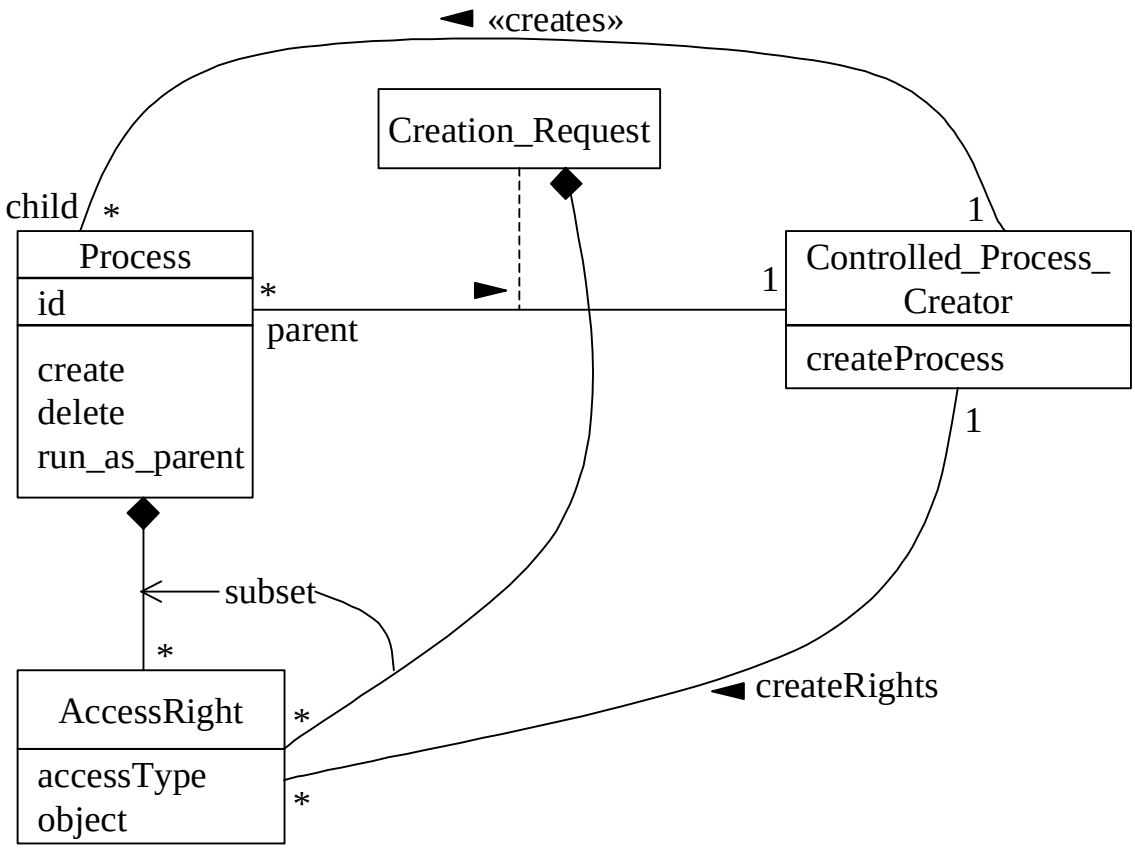

Figure 4.2 Process creation

4.4 Modes of execution and protection rings

At least two modes of operation are needed to have any security. Most hardware architectures use a *supervisor* and a *user* or *problem* mode. These modes define a coarse pair of domains. In the user mode some instructions, called *privileged* instructions, cannot be executed directly. Some of these instructions manipulate the state of the processes and clearly should not be accessible to users. Supervisor mode is reserved for the kernel and in this mode all the instructions can be executed. The state of a process is kept in a Program Status Word or Process Vector, which is part of the Process Control Block. When a user process needs to perform a system action, it issues a *system call* to the supervisor (typically by trying to execute a privileged instruction that traps to the operating system), which performs the action on its behalf and may check the parameters.

Some hardware architectures define a set of *protection rings* (typically 4 to 32) with hierarchical levels of trust [Fer08]. Rings are a generalization of the concept of mode of operation and define finer domains of execution. They have been used in Multics and the Intel processors among others. A combination (process, domain) corresponds to a row of the access matrix. Crossing of rings is done through *gates* that check the rights of the crossing process. A process calling a segment in a higher ring must go through a gate. The use of machine instructions can be made more precise with rings; for example, the Intel X86 series divides instructions into four levels:

- Privileged instructions that can only be executed at level 0 (the most privileged level).

- Instructions that can execute up to level 1.
- Instructions modified according to level.
- Instructions that can be executed at any level.

As indicated, in systems with two modes, user processes can directly call the supervisor through system calls. This is risky because a typical OS has many system calls and an attacker can take advantage of those that do not perform complete checks. A better approach is possible when using rings [Rin]. In this case, processes are assigned to rings based on their level of trust; for example, if we had four rings we could assign them in decreasing order of privilege to: supervisor, utilities, trusted user programs, untrusted user programs. Figure 4.3 shows a typical use of rings. The Intel X86 architecture applies two rules:

- Calls are allowed only in a more privileged direction, with possible restriction of a minimum calling level.
- Data in a segment at level p can be accessed only by a program executing at a more privileged level (<= p).

The Program Status word of the process indicates its current ring and data descriptors also indicate their assigned rings. These values are compared to apply the transfer rules.

Ring crossing applies protected entry points. A process calling a higher-privilege process can only enter this process at pre-designed entry points with controlled parameters. As indicated earlier this prevents bypassing entry checks and may protect against some attacks.

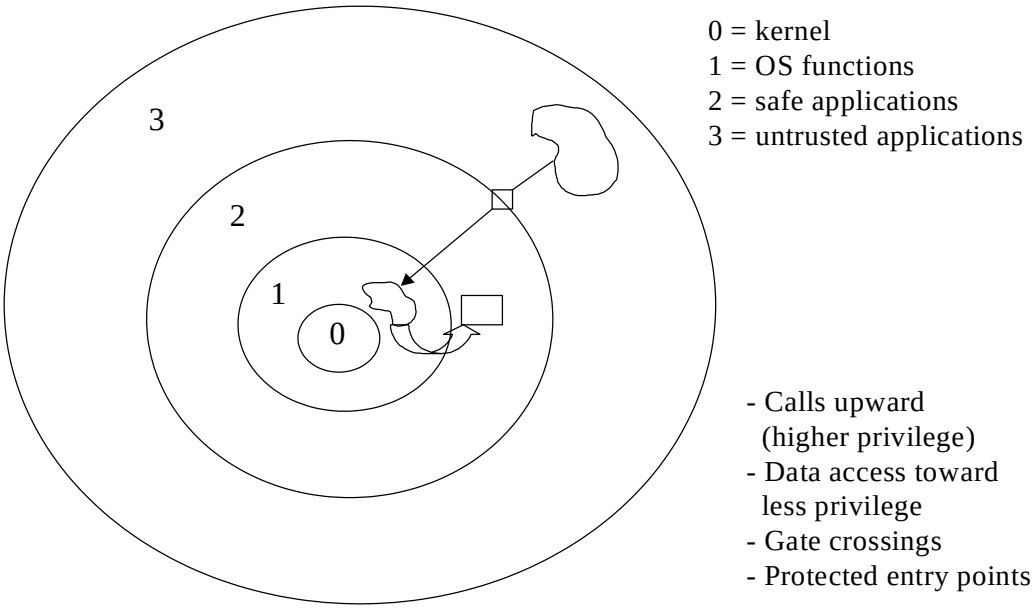

Figure 4.3 Privilege rings

Another possibility to improve security is to allow calls only within a range of rings; in other words jumping many rings is suspicious. Multics defined a *call bracket*, where calls are allowed only within rings in the bracket. More precisely, for a call from procedure i to a procedure with bracket (n1, n2, n3) the following rules apply: if n2<i<=n3 the call is allowed to specific entry points; if i>n3 the call is not allowed, if i < n1 any entry point is valid. This extension only makes sense for systems that have many rings.

Rings don't need to be strictly hierarchic, partial orders are possible and convenient for some applications. For example, a system including a secure database system could assign a level to this database equal but separated from system utilities; the highest level is for the kernel and the lowest level is for user programs. This was done in a design involving an IBM 370 [Fer78].

In IBM's AIX and Windows NT (and later versions), resources are represented by objects and the access types can be any operation that applies to the type of resource, e.g., print document, open file. In standard Unix resources are reached through programs, embodied in executable files. The access types are just read/write/execute. Memory is a very important resource and we discuss it in a separate section.

4.5 Memory protection

In early machines memory areas were protected with locks and keys, an area of memory was assigned a lock value and a process was assigned a key that when matched gave it access to that area of memory. That approach was restrictive and inflexible, keys granted no access or full access, and some areas of memory were defined as read only. Locks were stored in registers assigned to fixed partitions. The situation was improved by using pairs of base/limit registers which indicated the boundaries of the memory area allocated to processes. The two registers indicate from where the running process can fetch instructions and the area of memory it can access, The address calculation part of the execution sequence of a typical instruction would be: if (PC) > L trap [(PC) indicates the contents of the Program Counter relative to the base, L is the limit] ; else MAR ← B+ (PC) (B is the base register, MAR is the Memory Address register). This approach is more flexible in that the pair of registers can point anywhere in memory but still it can only define full access or no access.

However, the need to support many processes and control different modes of access makes these approaches impractical except for small specialized systems. For memory allocation most systems use paging, segmentation, or a combination of both. When paging is used, a process has a page table that indicates the units of memory (pages) allocated to it. Page table entries can be used to describe the permitted type of access to pages, such as read, write, execute (Figure 4.4). Because pages do not correspond to logical units in a program this protection is rather imprecise. It is, however, the most commonly used approach, because of its simplicity and good performance. A better approach for security is segmentation, where processes are given a descriptor segment that contains segment descriptors that indicate their type of access to a set of memory segments (Figure 4.5). Segments are variable-size units that correspond to logical

program units, e.g., code, procedures, data. Systems that use segmentation can apply a more accurate protection. Two approaches to implement this idea are commonly used, capabilities and descriptors.

Capabilities are a generalization of base/limit register pairs that include the type of permitted access and which are given to the process, which keeps them for subsequent use. They may combine access rights and addressing or define just access rights for resources. In a general sense they can be considered implementations of the rows of the access matrix model. Their security depends on having a trusted process that assigns capabilities to processes and on the users' inability to modify them. Usually hardware assistance is needed to enforce these restrictions, capabilities are placed in special registers and manipulated with special instructions (Plessey P250), or they are stored in tagged areas of memory (IBM S/38). Capability systems only require trust on the capability handling mechanisms, a small hardware/software combination, which makes them in principle the most secure architectures possible. Because sets of capabilities can define very fine protection domains, we don't need to use rings. Typically capability systems use a single-level virtual address space (See Section 4.6) and all resources are mapped to this space, which makes them a way to control access to all resources, not just to memory areas. A problem is *revocation* of capabilities; this can be handled through indirection (P250) or by relocating objects in the virtual address space (IBM's S/38). Both approaches are rather complex and reduce performance. The need for specialized hardware and the difficulty of revoking them has prevented their popularity, although a few capability architectures have been developed commercially, such as the Plessey P250 [Ham73], IBM S/38, and IBM S/6000 [Cha90]. However, they have practically disappeared in the last years. While hardware support is important, it is possible to build capability-based operating systems that require minimal hardware support at the cost of some extra overhead [Fei79].

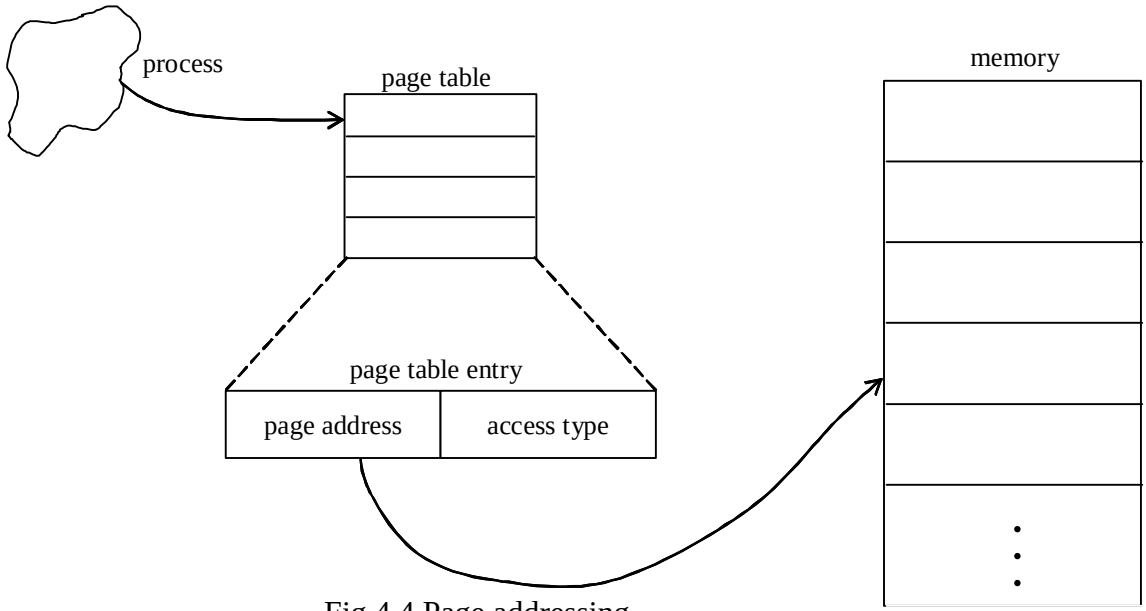

Fig 4.4 Page addressing

Descriptors are similar to capabilities but instead of being carried with the process code they are loaded when the process is created, i.e. a process is "lent" a set of descriptors to perform its job and these are taken back when the process ceases execution. Because the descriptors are used also for addressing they are handled by the memory allocation unit of the operating system and we need to trust that unit. The Intel X86 and Pentium series use descriptors for memory access control and to represent gates in their ring architectures.

Descriptors and capabilities are usually combined with address translation and then their use adds very little overhead, although ring crossing is relatively slow. The need for revocation mechanisms also slows down capability systems. A set of descriptors or capabilities when assigned to a process can be seen as embodiments of rows of the access matrix. Capabilities and descriptors are valuable not only for security, but also for reliability; the Plessey 250 used capabilities mostly to get high reliability [Ham73]. The set of all the pages or segments (described by page entries or descriptors) assigned to a process represent its *virtual address space (VAS)*; the set of all pages or segments that can be allocated to processes is the *system virtual address space.* Section 4.6 discusses the structure of these address spaces.

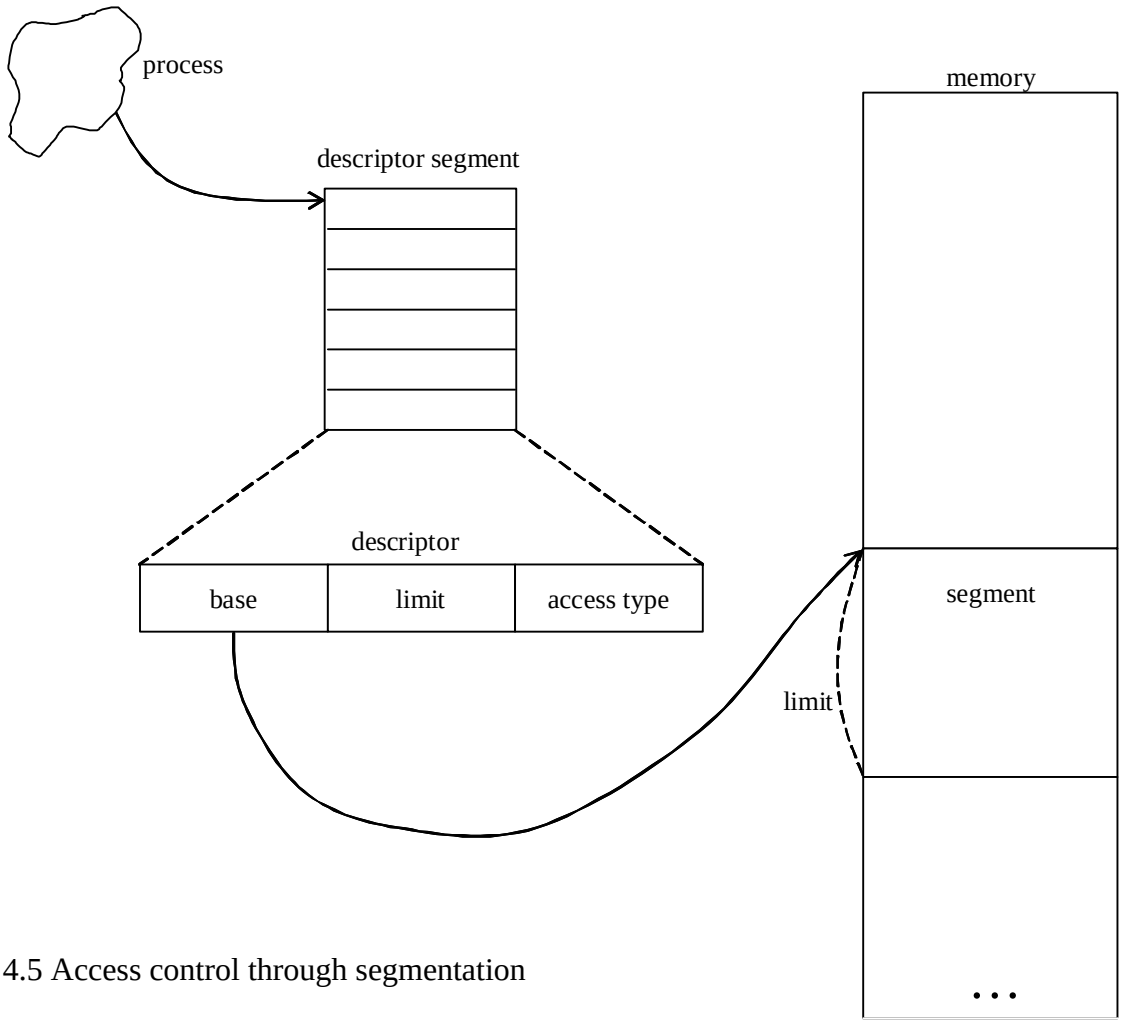

Fig 4.5 Access control through segmentation

Figure 4.6 shows the structure of an address space controlled through descriptors or capabilities, the Controlled Execution Environment pattern [Fer02]. As a process executes it creates one or more **Domains**. Domains can be recursively composed (an instance of the Composite pattern [Gam95]). The descriptors used in the process' domains are a subset of the **Authorizations** that the **Subject** has for some **Protection Objects** (defined by an instance of the Authorization pattern of Chapter 3). **ProtectionObject** is a superclass of the abstract **Resource** class and **ConcreteResource** defines a specific resource. Process requests go through a **ReferenceMonitor** that can check the domain descriptors for compliance.

Sometimes a process needs to execute in the domain of another process; for example, to clean up or perform some temporary job in that domain. For this purpose, some systems use the concept of *amplification.* A process sets the domain indicator of another process to let it run in its own domain. This idea was introduced for capability systems [Lev84]; in particular, it is used in Hydra and the iAPX432. These systems add rights to each capability as needed by the amplified functions. The idea has also been used in other type of systems, e.g., Unix (setuid).

Instead of segments some systems use *objects* as units of allocation. An object is made up of several segments that correspond to its data and operations sections. Examples of this approach are IBM's S/38 and S/6000, and operating systems such as AIX and Choices [Rus89]. If we use object-oriented languages, this VAS structure is close to the language structure and provides a finer protection.

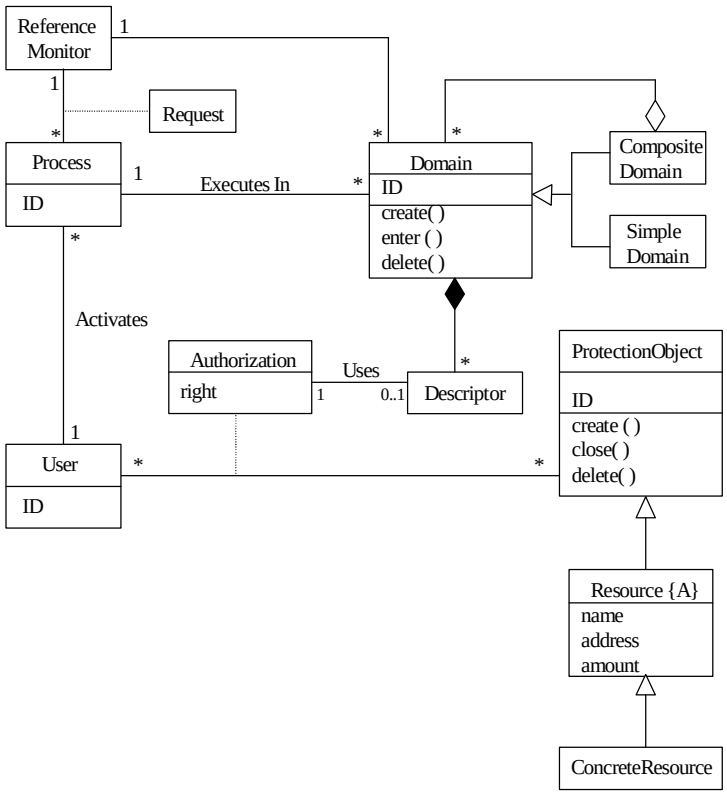

Figure 4.6 The Controlled Execution Environment pattern

4.6 Address space structure

Another important aspect of hardware security is the structure of the virtual address space presented by the hardware [Fer85]. There are four basic approaches (Figure 4.7);

- *One address space per process*. The supervisor gets its own address space. This is used, for example, in the NS32000, WE32100, and Clipper microprocessors. This approach provides good isolation but sharing is complex (special instructions to cross spaces are needed).
- *Two address spaces per process*. This is used in the Motorola 68000 series. Now 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. This approach has the problem of complex sharing plus it may result in a poor address space utilization.
- *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. This is not the best with respect to security (the supervisor has complete access to the user processes and it must be trusted) but it is convenient for sharing utilities and other system programs. Another disadvantage is that the address space available to each user process has now been halved.
- *A single-level address space*. Everything, including files, is mapped to this memory space. Multics, IBM S/38, IBM S/6000, and HP's PA-RISC use this approach. This is the most elegant solution (only one mechanism to protect memory and files), and potentially the most secure if capabilities are also used. However, it is hard to implement due to the large address space required [Kol92].

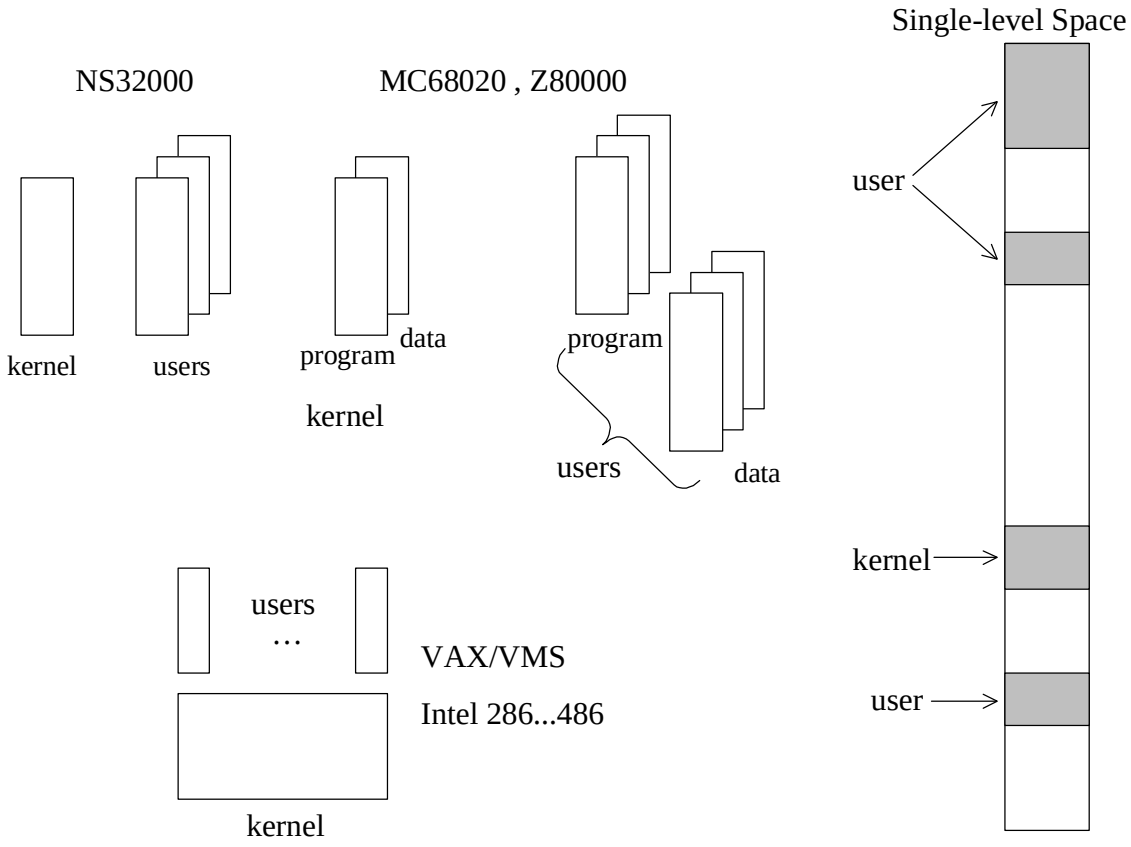

Figure 4.7 Virtual address space structures

4.7 System architectures of operating systems and their effect on security

Five architectures are typically used to build operating systems: monolithic or unstructured, modular, layered, virtual machines, and microkernels. Most practical systems use a combination of these architectures; for example, a virtual machine operating system can be layered. We discuss below their properties, focusing on security aspects. Patterns for them are described in [Fer05].

Monolithic architectures. This approach was used in earlier operating systems and is rarely seen now. The operating system consists of a collection of procedures, where each procedure provides a specific set of services. When they need services, the user processes call operating system procedures, which in turn may call other OS procedures. In its basic form there is no information hiding; every procedure interface is visible to all the other procedures. Some structure is possible by grouping the procedures into service and utility procedures [Tan97], where the utility procedures are in a lower level than the service procedures. The open interfaces provide more possibilities of attack and this architecture, while simple, is not secure. All the processes run in supervisor mode, which

also facilitates attacks. The lack of modularity doesn’t allow good use of concurrency, which may affect performance.

Modular architecture. This is an improvement of the monolithic kernel. In this approach the OS services are separated into modules, each representing a basic function or component. The core kernel is always in memory and can start itself and load modules. Whenever the services of any additional modules are required, the module loader loads the appropriate module. Each module performs a function and may take parameters. For example, a web browser uses an HTML renderer to display a webpage. In turn, the HTML renderer uses a jpg-renderer to display jpg images. Flexibility to add/remove functions contributes to security in that we can add new versions of modules with better security. Each core component is separate and talks to the others over known, possibly controlled, interfaces. It is possible to partially hide critical modules by loading them only when needed and removing them after use. By giving each executing module its own address space we can isolate processes. This approach is used by Solaris 10, ExtremeWare, and some versions of Unix (Free BSD), and Linux.

Layered architectures. In this approach the operating system is a hierarchy of layers of abstraction, where each layer presents an interface to the higher layers and hides the details of the lower layers. The lowest level layer is the kernel, which normally includes process creation, destruction, communication, and scheduling, memory management, and some basic I/O. The middle layer(s) includes file management and higher-level I/O. The highest level includes editors, user interfaces, mail systems, and other utilities (usually called system services). In general, the overall features and functionality of the OS are decomposed and assigned to hierarchical layers. This provides clearly defined interfaces between each section of the operating system and between user applications and the OS functions. Layer i uses services of a lower layer i-1 and does not know the existence of a higher layer i+1. Variations of this structure are used in Unix, Linux, and Windows. This architecture is another application of the Layers pattern, seen in Chapter 1. Its advantages include: Lower levels can be changed without affecting higher layers (we can add or remove security functions as needed). The use of clearly defined interfaces between each OS layer and the user applications can improve security. In addition, the fact that layers hide implementation aspects is useful for security in that possible attackers cannot exploit lower level details. All this comes at the cost of some performance overhead because of the need for extra calls to go across the layers.

Microkernel-based architectures. A microkernel includes mostly communication functions and a set of some fundamental services, typically functions such as process creation/deletion, process communication. All other non-essential functions are implemented as part of internal or external servers. A pattern for this architecture is shown in Figure 4.8 (based on [Bus96]). The client interfaces with the kernel through an Adapter. This routes the service request to the either the microkernel or to an external server. The microkernel class is used mostly to establish communication and to perform very basic functions. Additional functionality is decoupled into *internal servers*. *External servers* implement their view of the microkernel as seen through its interfaces. This decoupling makes the microkernel reusable and the operating system becomes highly extensible, we just need to add more internal or external servers. This approach is a type of client/server system where the microkernel provides a common facility between client programs and services. A microkernel brings the following issues for security:

- Flexibility and extensibility – if you need an additional function or an existing function with different security requirements you only need to add an external server. Extending the system capabilities or requirements only require addition or extension of internal servers.
- The Microkernel mediates all calls for services and can apply authorization checks. In fact, the microkernel is in effect, a concrete realization of a reference monitor.
- The well-defined interfaces between servers allow each serve to check each request for their services and to check their code if needed.
- Can add even more security by putting fundamental functions in internal servers.
- Servers usually run in user mode, which further increases security.
- The microkernel is very small and simple and can be verified or checked for security.
- It is harder to attack the kernel itself.
- A possible disadvantage is the extra overhead due to indirection.

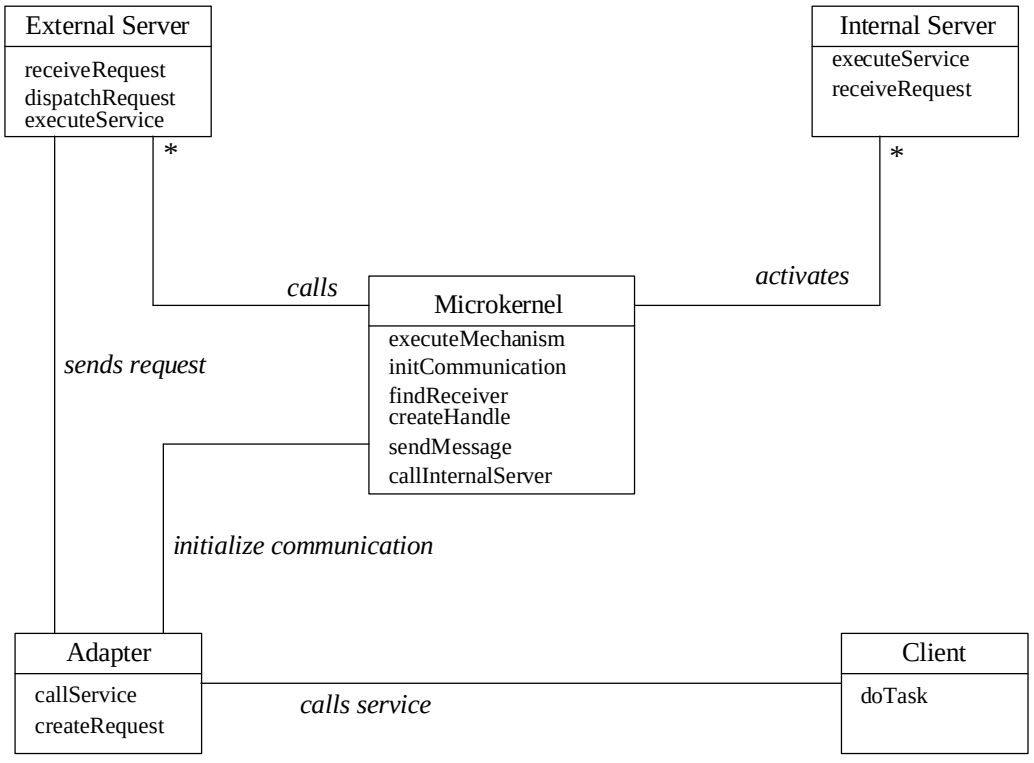

Figure 4.8 The Microkernel pattern

Virtual machine (VM) operating system. The idea here is to apply multiprogramming at the hardware level. Each VM runs in a virtual copy of the hardware implemented using a *Virtual Machine Monitor* (VMM), that acts as a specialized kernel for the complete system and is the only one with access to the real hardware. The VMM creates multiple replicas (virtual machines) of an instruction set architecture on one real system and allows one executing environment to run different operating systems (Figure 4.9). The VMM keeps tables describing the specific characteristics of the operating systems that run in the virtual machines so it can interpret system calls. Privileged instructions in the VMs are intercepted by the VMM, which interprets them according to the operating system running in the corresponding VM. Their value for security is based on their isolation properties. Microkernels can be used to implement VMMs.

Virtual machine OSs were invented at IBM in 1970 and they disappeared for a while. A project between SDC and IBM in the late 70s produced a Kernelized Virtual Machine (KVM/370) that used the VMM to separate virtual machines that use levels according to the Bell-LaPadula model [Gol84]. Virtual machines are making a comeback, several institutions have now research projects in this area [nps], a company (VMware) was started in 1998 [vmw], and IBM worked with it to develop a system to let multiple virtual servers to run in one IBM server. Similarly, Microsoft acquired Connectix for its own virtual machine development [con]. VM operating systems are the basis of cloud computing, which virtualize not just the processor but whole platforms, including operating systems and middleware. Cloud computing has made this architecture become very important.

Intel has been working on a project called Vanderpool. It's a virtualization technology to be introduced in its processors, supported by "VMX" instructions. It looks similar to the compartments idea in Virtual Vault and is supposed to be important for distributed processes, web services, gateways, etc., especially since it could be used in their processors [Ivt].

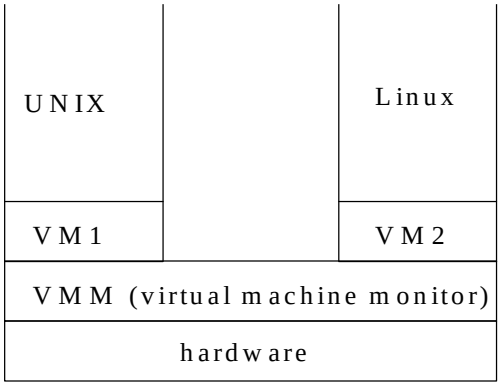

Figure 4.9 A virtual machine operating system

Security advantages of this concept include:

- The VMM intercepts and checks all system calls. The VMM is in effect a Reference Monitor and provides total mediation on the use of the hardware. This can provide a strong isolation between virtual machines [Ros05].
- The VMM is small and simple and can be checked for security
- There is a well-defined interface between the VMM and the virtual machines.
- It is possible to move operating system security mechanisms, e.g. intrusion detection, outside the operating system to protect them from attacks [Ros95].
- It is possible to monitor and control the network connections of the virtual machines to assure that they are protected and not malicious. The VMs can also be authenticated to the network [Ros05].
- Isolating the software stacks of applications in separate machines simplifies reasoning about their security because they are free from interference from other applications [Ros05].
- Each environment (VM) does not know about the other VM(s), this helps prevent cross-VM attacks.

Other architectures have been proposed but none has been widely used. An interesting proposal is to use a separate security module or subsystem to perform security functions for both the operating system and the database system [Spo84]. Some experimental systems have used *reflection* to provide adaptability and security [Son94]. A reflective system uses metaclasses whose objects handle different adaptability aspects, such as security, checkpoints, or fault tolerance.

4.8 File protection

There are two basic approaches for protecting files:

- Map files to a single-level virtual address space (VAS), as indicated in Section 4.6. Files are then protected by capabilities or descriptors.
- Have a special file authorization system. This is done in most commercial operating systems.

In the latter case, special “file permissions” are defined and access is decided by looking at these file permissions for a match. Section 4.10 describes file permissions for Unix. The search order for permissions is important, especially if there are implied access rights (rights that are implied by other rights). Permissions may be implemented as Access Control Lists associated with the file.

Most file systems use the concept of *owner*, the creator of a file. The owner has full rights for the files she creates, including the right to delegate rights to other users. This is fine for private files but it is not a good idea for files that contain institution data which is not owned by any user.

Another concept for accessing files is the use of groups. Users in a group may get access to a file from its owner. The group that get the rights may be the group of the owner, the group of the creating process, or defined in other ways.

Files are organized in directories that provide a search path for them. Permissions for directories have a different meaning than permissions for files. Section 4.11 shows the interpretation used in Unix.

Figure 4.10 shows the *File Authorization pattern* [Fer01]. This pattern describes a file structure where the file components may be simple files or directories that may contain files or other directories (the *Composite* pattern [Gam95]). The subjects (users, groups, or roles) may have authorizations to use specific workstations and to access file components (access permissions). Figure 4.11 shows a sequence diagram to write to a file after opening it. The open operation sets up the required authorizations that will be active during the use of the file.

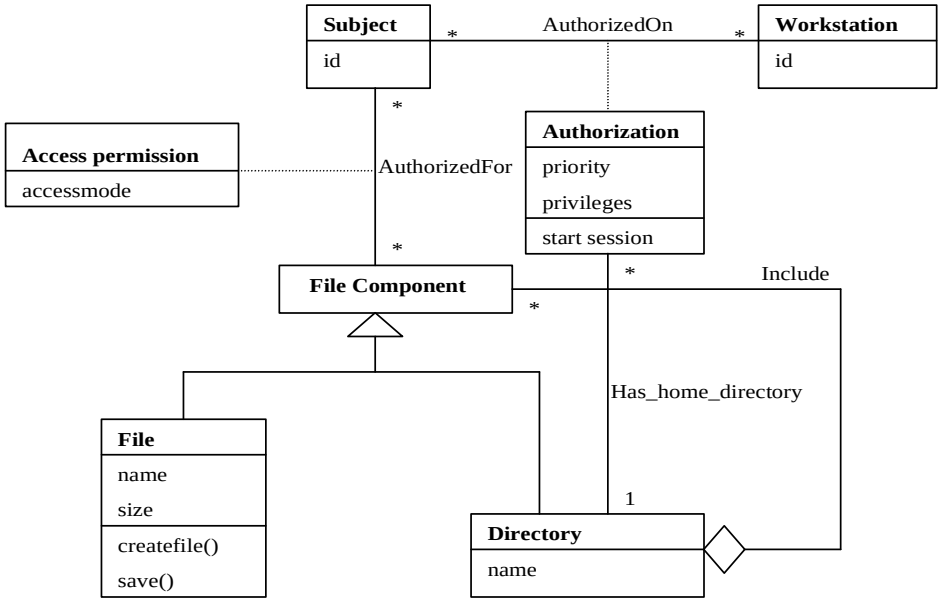

Figure 4.10 The File Authorization pattern

4.9 User and system authentication

Before they can perform any activities in the system both users and systems must identify themselves and be authenticated. *Identification and Authentication* (I&A) use some kind of protocol to establish identity. I&A is the basis for authorization and for logging, it provides *accountability*. In Figure 4.12 a user attempts to login providing his identity. The system performs a protocol to verify this identity using some authentication information. Once verified, the system may provide a proof of authentication and a

handle (reference) to the user's account. For example, IBM's z/OS provides an *accessor environment element (ACEE)*, which follows the authenticated process during its execution [Gus01].

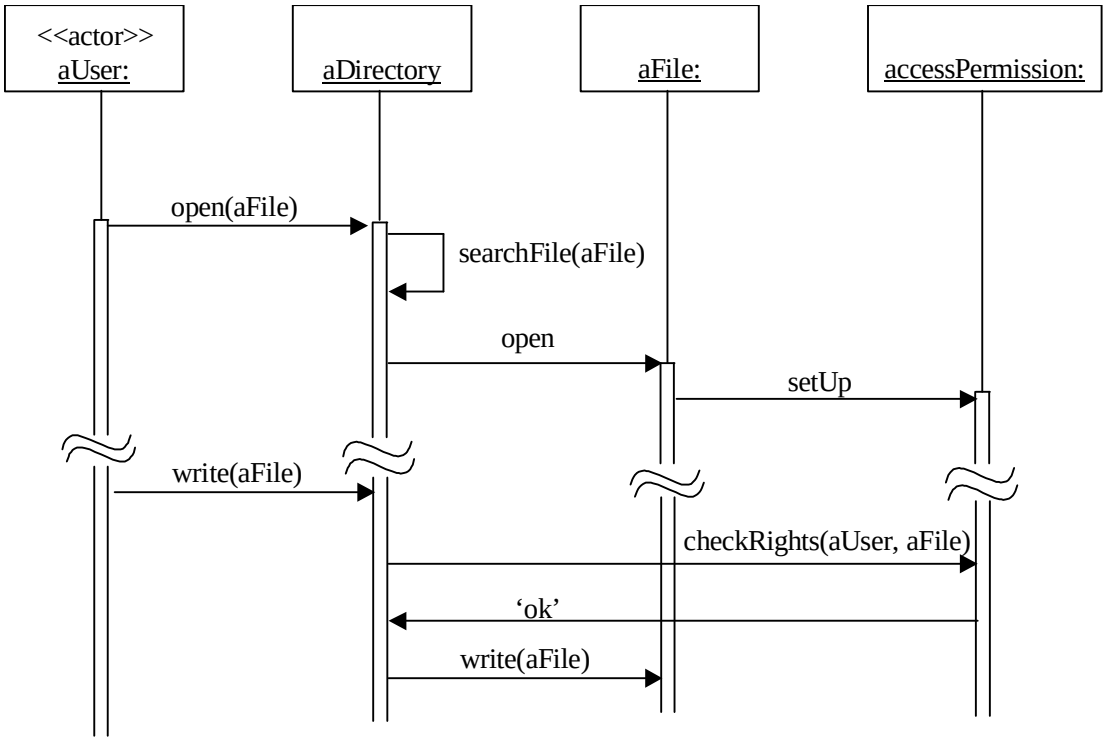

Figure 4.11 Sequence diagram for writing to a file

Passwords are considered a relatively weak approach. Most passwords are easy to find using a good dictionary, educated guesses, and a program that tries them in succession. For better security the Unix password system uses slower encryption, produces less predictable passwords with a password generator, and uses 'salting' [Mor79]. Salting implies that the password is encrypted using the concatenation of the time of the day and the process id as a key; the encrypted password is then concatenated with the time of the day and the process id. The main problems with passwords are that users choose poor passwords and that they are kept for long periods of time, which means that the attacker has time to find a way to discover them. A common attack is to get a copy of the encrypted passwords (some systems don't protect the password file itself) and perform dictionary or exhaustive attacks.

More secure (and costly) approaches for user authentication use:

- *Smart cards*, where a card with some processing power is used to establish the connection. Smart cards use a PIN provided by the user and other information such as date, time, or name of the user to send an encrypted message to the system (typically some Authentication Server). Only if the system can decipher the message the user is connected. The card must be tamper-resistant to prevent a person who steals the card from finding the PIN in the card. Given that these cards have a significant amount of memory there have been proposals about including in them financial, health, and employment information about their owners, although this may bring additional privacy problems [She02].
- *Biometrics*. This is the use of some unique personal characteristic, e.g., fingerprints, retina scan, face contour, face recognition, signature dynamics [Jai00]. This can be a more secure approach but it is more expensive and slower than the other two [Mil94]. Fingerprint recognition security has been put in doubt, people leave their fingerprints in many places and gummy fingers appear to fool most systems [Mat02]. After September 11 this approach has become considerably more popular. This popularity has also been helped by the lower cost of some of these approaches.

Authentication between network nodes must use some protocol as discussed in Chapter 7. Authentication is an important problem for web services and wireless devices, we'll discuss these in Chapter 8.

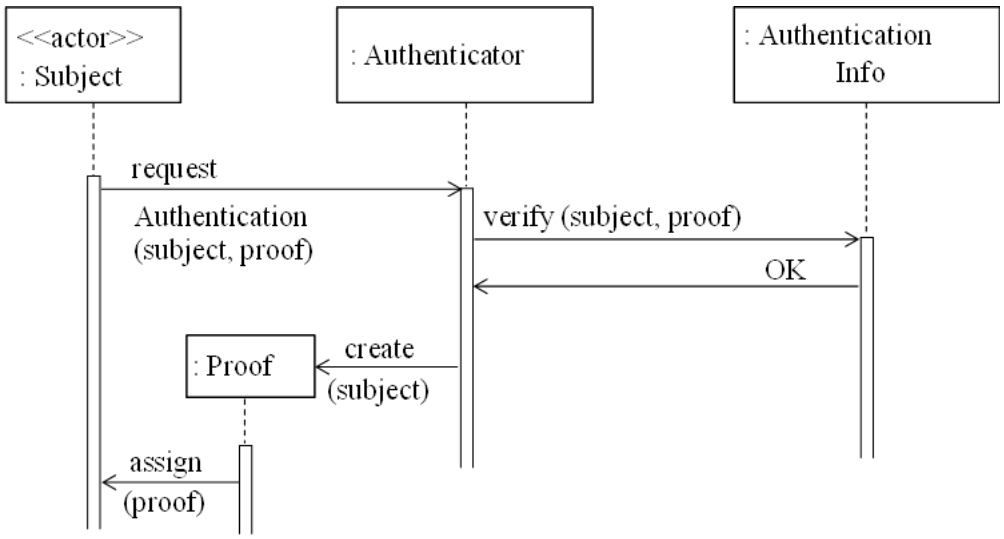

Figure 4.12 Sequence diagram for use case "Authenticate a user"

4.10 Authorization and access control

Some systems, typically mainframes, use a separate authorization service that contains ACLs that define rules to control access to resources (files, data sets, programs, workstations). When a process requests access to a resource, the resource manager invokes the authorization service to decide access. The advantage of this approach is that

it is possible to change the authorization structure without affecting the rest of the operating system. Its disadvantage is some extra overhead. An example of this approach is the z/OS of IBM, which uses the Resource Access Control Facility (RACF) to evaluate access [Gur01]. In other systems, e.g. Windows, the authorization system is integrated with the rest of the functions (Section 4.11).

4.11 Commercial operating systems

There are relatively few types of operating systems used in commercial systems, for small and medium systems most of them are variants of Windows or Unix. Servers and mainframes add a few varieties such as z/OS, while portable devices have a larger variety but still only a few are used (Windows, Symbian, PalmOS, and Linux). A comparison of Unix and Windows security is given in [Vie00]. Their conclusion is that neither is clearly superior in this respect. An analysis of the security of Windows NT is in [Mud97], who considers it rather poor, and in [Hya], who considers it a good improvement over "other systems". Windows 2000 introduced several improvements over Windows NT but we know of no study comparing it to other operating systems. Based on actual incidents the Microsoft operating systems appear weak in spite of their relatively good security architecture; it seems that most of the problems are due to poor implementation and unnecessary complexity (Windows 2000 has over 40 millions lines of code). Another problem is their lack of clear interfaces between modules; they claim that their browser, media player, and other plug-ins cannot be separated from the basic operating system. Unix appears to be more secure, although it is still far from being a high-security system. Linux, although following the general security architecture of Unix, appears relatively better, mainly because its open source code has been carefully studied by many and because of its relative simplicity. This should make it more secure [lin]; in fact, it has a smaller rate of reported incidents than other versions of Unix and much smaller than Windows. We will see in Section 4.13 that there are some hardened versions of these systems. Operating systems for portable devices are considered later. Standards may help to select operating systems but none exists although IEEE is working on one [iee].

Unix

There are several versions of Unix, the original ATT was followed by others such as System V, BSD, HP-UX, AIX, Solaris, etc. What we describe below is valid for all of them unless otherwise indicated. Linux follows the general security structure of Unix, but it is different from the others for being open source (actually, some versions of Unix such as BSD are also open source).

In Unix every user is given a unique identifier, UID. If a user creates a file, her UID is associated with the file and the user becomes its *owner*. Users are divided into groups and each group has a unique group identifier, GID.

A set of 12 bits (a *file permission*) defines how the file may be accessed by the owner, by the group to which the owner or process belongs, and by the rest of the users (world). There is no way to specify access for specific users, i.e., a file permission is a restricted form of the access matrix. A permission defines only three types of access for files: read, write, and execute (these are interpreted differently for directories). A permission is a

four-digit octal number (three bits define the owner's rights, three bits define group rights, three bits define everybody else's rights) and can be set by a special instruction *chmod*. The permission includes also three bits that have the following purpose:

- A setuid bit, that lets a process run with the id (and the rights) of the owner of the file (a type of amplification).
- A setguid bit, that does the same with respect to group rights (the group of the creating process in System V or the directory group in BSD).
- A sticky bit, that is used for memory management (to keep the program text in memory after the process completes execution). This bit here is an example of a violation of the "least-common mechanism" principle (See Chapter 2)..

For directories, execute right indicates search permission, read right gives access to the file names in the directory, and write allows modification of directory entries. Combining file and directory rights allows a user to perform some illegal or unintended actions [Sum97].

Unix files use descriptors, called *inodes,* to hold access permissions. The inode for a file is brought into memory when the file is opened. A PATH environment variable defines the order in which a requested file is to be found. The access evaluation algorithm (to decide if a request is authorized) is very simple; the system checks first for owner rights, then for group rights, then for world rights.

Unix uses a mixture of user and role to assign rights. Its file system uses users and groups as subjects but some system functions are special roles with their own rights, e.g., root or superuser, daemon, agent, guest, ftp, uucp [Gar96].

In Section 4.11 we discuss several security weaknesses of Unix. The different implementations of Unix have different levels of security, depending on how they have plugged the original holes. Some, like the HP-UX 11i, have a trusted mode, with additional constraints: protected password database, auditing, terminal and serial port restrictions, password generation, and others. BSD 4.1 added a variety of security improvements. Vendors may have also a hardened version of their main version of Unix, with enhanced security, as we discuss in Section 4.12.

Windows

Windows has a more elaborate access control structure and allows more types of access to files than Unix [Hya]. Its subjects are users, groups, and other computers and have unique identities. After a subject has been authenticated the system keeps an *access token* that contains information about the subject rights in the form of security attributes. There is also a mechanism similar to setuid in Unix, the *impersonation token*, where a subject can use another subject's access rights. File rights include:

- *read* access to a file that allows reading the data in the file and execution of a file,
- *no-access* right, used for negative authorizations,
- *change* access that allows file modification and deletion,
- *full* control access, that allows ability to modify file permissions and to transfer ownership of a file, i.e., this is an administrative right.

As indicated earlier, Windows uses objects as units of resource access. An object includes a *Security Descriptor* that contains [Cal00]:

- The Owner Security Identifier (SID), which can be a user, a group, or a computer. These are the subjects, called 'principals' by this sytem (and by several others).
- The group SID.
- A DACL (discretionary access control list), indicating which SIDs can access the object and how.
- A SACL , System access Control List, used for auditing.

Note that both NT and 2000 use the concept of Owner, which is the most powerful principal and who can change permissions. The DACL indicates that these systems implement a full access matrix model.

There is a Security Reference Monitor that consults the corresponding ACL when a process requests an access. Entries in the ACL may be positive or negative; this allows configuring open and closed systems.

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. Passwords are encrypted using the DES algorithm and stored in encrypted form. Windows NT reached the C2 DoD (Orange Book) level.

Windows 2000 introduced the concept of *Active Directory* (AD) that provides an administrative structure for distributed domains. The AD is a tree that acts as a repository or registry for all resource information. A *domain* is a group of systems that share the same directory; in this sense the AD is a directory of directories (the Composite pattern again !). Each domain has a unique name and provides centralized administration for user and group accounts. Users can cross domains after being properly authenticated. The AD introduces the concept of *dynamic inheritance* of rights along the tree. This is a direct application of the concept of *implied authorization* that we proposed in 1975 [Fer75]. Implied access rights are very convenient for system administration because the administrator can understand better the structure of rights in the system. W2000 also improved and expanded the use of groups for security and introduced the use of Kerberos for distributed authentication (See Chapter 7).

The latest version of Windows is XP. A Service Pack, SP2, intended mostly for security was released in 2004. It is not considered a big improvement [Gre04]. Among its flaws it includes a firewall that blocks only input traffic and many of its security settings are just status indicators.

4.12 Weaknesses of commercial operating systems

There are many vulnerabilities in commercial systems that have been exploited by hackers in many ways; in fact, they have been used for most Internet attacks. Operating systems are very complex systems and most of the times they are built in an ad hoc manner, without using systematic secure software development processes; this makes the

occurrence of flaws a likely event. Vendors include many functions, which are hard to configure and may be avenues for attacks.

We enumerate below some of the most common flaws of commercial operating systems. Some of these have been corrected in current versions of them, but it is clear that enough flaws remain because the attacks continue. Some of these flaws are:

Authentication problems--Both Unix and Windows use passwords for authentication. Unix keeps passwords encrypted but the password file is readable by all users. This allows a user to make a copy and use dictionaries and parallel processing to guess passwords. The file used to store passwords (etc/password) includes also user information, readable through the **finger** command. This information is useful to hackers to make a better guess of passwords. Some systems use a protected shadow file to correct this problem.

Few modes of operation--In these systems process protection is based mainly on the user/ supervisor mode separation and kernel processes are not protected against each other. Even if hardware architectures offer further protection, e.g., descriptors and rings, commercial operating systems usually do not use them in an effort to get more performance.

The concept of superuser—This is an almighty user, typically intended for the systems administrator. This concept is a source of problems because it goes against the policy of least privilege. When this account is compromised the hacker can reach anything in the system as well as being able to erase his tracks. Newer systems have corrected this problem as discussed below.

Inheritance of rights –Most systems let forked processes inherit the rights of their parents. This is another flaw commonly exploited in attacks. If an attacker tricks a program in superuser mode to execute a Trojan Horse, this inherits the rights of that program and runs in superuser mode.

Transfer of rights between processes—In Unix if a bit in a file permission (*setuid*) for a file containing an executable program is turned on, the process executing that program acquires the rights of the file owner. Windows has an impersonation token, which has a similar effect. This is useful; for example, to let a user change her password (to let the user process write into the password file), but it violates the principle of accountability. Specifically, a user might manage to leave around a program with superuser rights and this bit turned on.

Lack of conceptual security model -- The file permission structure in Unix doesn't follow the access matrix or any of the classical security models. The interpretation of rights for directories makes things even more muddled; for example, a user with search and write access to a directory can delete all the files in a directory without having direct rights to any of them.

Directory problems—Directories made be made public for convenience but only search should be permitted in them. An attacker can place his own file in the path of a writable directory and maybe get higher privileges when the file is invoked.

Lack the concept of a trusted path-- A trusted path is a user connection to a part of the system that provides secure login, authentication, and rights [Los00].

Lack of audit facilities--Some systems do not have auditing facilities or the audit log is within reach of the superuser (and could be changed by a hacker acting as a superuser).
Complex, poorly designed, and poorly tested utilities-- Microsoft's Outlook is known to be full holes that have been exploited many times. The Sendmail program in Unix is another source of trouble.

Buffer overflow-- Buffer overflow occurs when a data structure in a procedure is filled with more values that it can hold. The overflow can be made to overwrite the return address and if the hacker put her code there her program could get superuser mode. Correction requires bounds check and this takes time, which explains the reluctance of vendors to correct this problem. In [Vie00] the authors say that this is a C problem (no check for buffer boundaries) and not the fault of the OS. This is true, however, there is no reason for OS vendors not to use a better language or to modify the run-time system to perform these checks; Microsoft changed the Virtual Machine of Java to suit marketing purposes, why not change C? This problem was first reported in 1968, so there is no way to pretend ignorance either. We discuss this problem in more detail in Chapter 5.

Denial of service-- This is a problem produced by a user or users hogging system resources. None of the known models for security can address this problem; Gligor defined necessary and sufficient conditions for this problem to be possible [Gli84]. Most operating systems put limits on the number of threads a user may have and on the number of files that can be opened but most systems do not count how many files a given user has created [Fil86, Mud97]. A bad-intentioned user can cause considerable disruption to the other users. Later we'll discuss the distributed form of this problem (a network problem originated because of operating system and protocol weaknesses).

Complex configuration and functionality-- These systems are complex and administrators make mistakes in configuring them. There are many unnecessary functions, demo programs, and rarely-used utilities which can be exploited by hackers. This is even truer for PCs where the users usually have no idea what they get in their software packages.

In summary, most operating systems have a variety of security flaws. Some are just implementation problems and are easily correctable, others are design flaws and much harder to remove. In Section 4.13 we look at some "hardened" operating systems where these errors have been corrected to different degrees.

4.13 Secure (trusted) operating systems

In the 1970's there was considerable work on trusted operating systems. These implemented multilevel models and their commercial versions were not successful. There is now renewed interest on them and several commercial versions have appeared that are more flexible than pure multilevel systems [Mil01]. The technology to build very secure operating systems appears to be available now [Har02].

Hardened versions of commercial systems

These try to improve security by reducing the known exposures of standard operating systems. For example: Administrative privileges can be reduced or divided into several groups or roles. Inheritance of rights in forked processes can be eliminated or reduced. Of the commercial systems a few have reached the E3/F-C2 classification of the Orange Book: AIX 4.3 and Windows NT Server and Workstation 4.0 (maybe others). To reach a C classification the system must implement an access matrix DAC model. Several systems have extended or are extending their authorization systems to comply, e.g. Linux [tru], and some of the systems described below.

IBM's AIX [Cam90]—AIX is a version of Unix that implements a TCB to support DAC. The kernel runs in system mode and it is part of the TCB. Some programs can run with high privilege and are considered trusted programs. Instead of read/write/execute rights Security policies are divided in three groups: access control, accountability, and administration. AIX defines an Abstract Data Type (class), with higher-level operations, appropriate for the type of object such as copy, save, query, and set. These accesses define an access matrix implemented as Access Control Lists. The ACLs are set by the owners of files and by administrators. ACLs can be permissive or restrictive. User identification attributes include user IDs, account audit classes, and group membership. User authentication is performed through passwords. Any security-relevant event is auditable. AIX reduces the privileges of the system administrator by defining five partially-ordered roles: Superuser, Security Administrator, Auditor, Resource Administrator, and Operator. A 64-bit version of AIX (version 4.3) reached the ITSEC rating E3/F2-C2.

.

Virtual Vault [HP, Rub94]—This is a secure platform for Internet applications that includes a trusted HP-UX operating system (A Unix variant), and firewalls and other protection devices. It is part of a secure server system, the HP Praesidium family of products. It uses compartments based on the multilevel model to isolate portions of the OS. It also reduces the root privileges and controls inheritance of rights in forked threads. According to HP, Virtual Vault is used in many banks around the world.

Trusted Solaris []---This is an extension of Solaris 8 that reaches the B1 level. It implements a MAC using labels and a DAC using ACLs and the concept of file owners, RBAC is used to restrict the rights of administrators, there is no superuser with all power. Regular users are controlled with Unix file permissions. They also use the concept of trusted path and do extensive logging. Trusted Solaris incorporates some protection between kernel units.

REMUS (Reference Monitor for Unix Systems) [Ber02]---This is an extension of Unix based on analysis of the vulnerabilities of its system calls and improving their use. This is an interesting approach that deserves further study.

Argus Pitbull [Arg]—This is a system based on:

- Compartmentalization based on a multilevel MAC model. It separates users and data by classifications and compartments.
- Least privilege applied to all processes, including the superuser. The superuser is implemented using three roles: Systems Security Officer, System Administrator, and System Operator.
- Kernel-level enforcement.

The system is applied as a layer on top of another OS, e.g., Solaris or AIX.

Trusted Linux varieties— An attempt to convert its permission system into a complete access matrix is the Linux trustees project [tru]. The NSA is sponsoring the development of a security-enhanced Linux [NSA], which uses a multilevel model for its kernel. Another project to fortify Linux was done at the HP's Bristol, UK, Labs. They built Trusted Linux, a system with a security kernel using mandatory controls based on labels intended to separate applications [Dal01]. The SuSE Linux Enterprise Server has achieved a Common Criteria Evaluation Assurance Level certification. Security Enhanced Linux is a project at the University of Utah [Spe].

Other trusted operating systems—Harris in Fort Lauderdale (later CyberGuard) used to market a multilevel version of Unix, the CX/SX (Night Hawk), based on UC Berkeley's System V. This system used labels and supported 256 levels and 1024 categories. It got a B1 rating. A C2 Unix system, the UTX/32S, was produced at Gould Inc. also in Fort Lauderdale, around 1985. It used "restricted-environment domains" for users and eliminated setuid and setgid.

Instead of extending or hardening existing systems it is possible to look for more fundamental ways to build secure operating systems. While the compatibility problem may hinder acceptance of such systems, in the long run they may become important. An interesting capability-based operating system is EROS [Sha02]. This system attempts to follow all the Saltzer-Schroeder principles, plus a few more of its own. Another objective is to formally verify some of its basic security mechanisms. Early work on secure kernels showed the value of capability systems and the need to start with small kernels that can be verified [Fei79]. Nizza is a small secure platform intended for applications with high security requirements. This architecture allows the coexistence of high security applications with legacy (less secure) applications running on Linux, an important requirement for any practical system [Har02].

4.14 Trusted hardware

Figure 4.13 shows a model of a Trusted Platform Module.

Microsoft also is intending to apply security through hardware, in its Palladium arch., see http://www.microsoft.com/presspass/features/2002/jul02/0724palladiumwp.asp

There was a paper about Palladium in IEEE Computer of July 2003: http://download.microsoft.com/download/c/8/0/c80ea683-9900-46ff-9c67-d7f14b0d3787/trusted_open_platform_ieee.pdf

Trusted Computing Group, http://www.trustedcomputinggroup.org

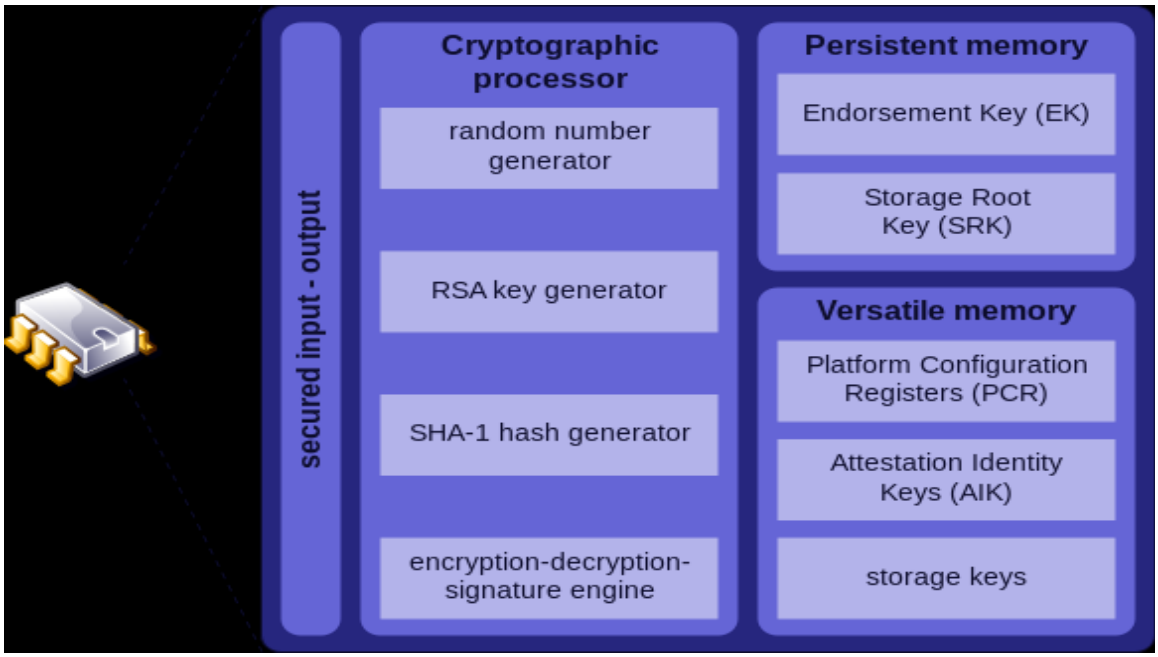

Figure 4.13. Architecture of a TPM

Exercises

4.1 Discuss the use of locks and keys to protect memory. Is this a convenient approach for modern operating systems? Do you think it could be useful for specialized systems?

4.2 Draw a sequence diagram for ring crossing in an architecture such as Intel's. Assume that there are gate descriptors that are invoked to make calls from lower to higher privilege rings.

4.3 What mechanisms in the operating system could stop or mitigate the effect of worms and viruses?

4.4 Discuss in detail the security advantages and liabilities when using a Modular OS architecture.

4.5 Discuss the security advantages of a layered microkernel. This is a microkernel operating system where the different functions (servers) are assigned to a hierarchical set of layers.

4.6 The Virtual machine pattern in the notes and in class considers only one processor shared by several virtual machines. In order to provide scalability, cloud systems need the virtual machines to be supported by several processors (a multiprocessor). Discuss the additional security problems introduced by this change in the hardware.

4.7 The security architecture of Windows is clearly better than the one of Unix. Why is it then that Windows get attacked in more different ways than Unix?

4.8 Compare Trusted Solaris to Argus from a security point of view.

4.9 Study the file structure of Windows. Does it match the File Access Control pattern?

4.10 Study the execution environment of Java 1.2. Does it match the Controlled Execution Environment pattern?

4.11 Consider the virus/worms examples of Chapter 1. What security mechanisms in the operating system could help to stop or reduce their effect?

4.12 How would you certify that a voting machine is secure? Because of the importance of the application this certification will require using redundant ways to make sure that nothing is left out (an application of the "Defense in depth" principle). The vote issued by a voter must be faithfully recorded and added to the counts while preserving the secrecy of the vote.

4.1 What would be the effect of using protection rings to stop or mitigate some malware, such as the Bagle worm?

References

[Ame83] S.R.Ames, M. Gasser, and R.R.Schell, “Security kernel design and implementation: An introduction”, *Computer*, July 1983, 14-22.

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

[Ber02] M. Bernaschi, E. Gabrielli, and L.V. Mancini, „REMUS: A security-enhanced operating system“, *ACM Trans. on Info. and Sys. Security*, Vol. 5, No 1, February 2002, 36-61.

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal., *Pattern-*

oriented software architecture, Wiley 1996.

[Cal00] M. Calbucci, “Windows 2000 Security Descriptors”, *Dr. Dobbs Journal*, November 2000, 57-66.

[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.

[Cha90] A. Chang, M.F.Mergen, R.K.Rader, J.A.Roberts, and S.L.Porter, “Evolution of storage facilities in AIX Version 3 for RISC System/6000 processors”, *IBM Journal of Research and Dev.*, vol. 34, No 1, January 1990, 105-110.

[con] Connectix Corp., *The technology of virtual machines*, white paper, 2001.

[Dal01] C. Dalton and T.H. Choo, “An operating system approach to securing services”, *Comm. of the ACM*, vol. 44, No. 2, February 2001, 58-64.

[Del07] N. Delessy, E.B.Fernandez, and M.M. Larrondo-Petrie, "A pattern language for identity management", Procs. of the 2nd IEEE Int. Multiconference on Computing in the Global Information Technology (ICCGI 2007), March 4-9, Guadeloupe, French Caribbean. http://www.computer.org/portal/web/csdl/doi/10.1109/ICCGI.2007.5

[Fei79] R.J.Feiertag and P.G.Neumann, “The foundations of a provably secure operating system (PSOS)”, *Procs. of AFIPS 48*, 1979, 329-334.

[Fer75] E. B. Fernandez, R. C. Summers and T. Lang, ``Definition and Evaluation of Access Rules in Data Management Systems," *Proceedings 1st International Conference on Very Large Databases*, Boston, 1975, 268-285.

[Fer78] E.B.Fernandez, R.C. Summers, T. Lang, and C.D.Coleman, “Architectural support for system protection and database security”, *IEEE Trans. on Computers*, vol. C-27, No. 8, August 1978, 767-771.

[Fer85] E.B.Fernandez, “Microprocessor architecture: The 32-bit generation”, *VLSI Systems Design*, October 1985, 34-44.

[Fer01] E.B.Fernandez and R. Pan, “A pattern language for security models”, *Procs. of PloP 2001*, http://jerry.cs.uiuc.edu/~plop/plop2001

[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 systems access control”, *Procs. of EuroPLoP 2003*, http://hillside.net/europlop

[Fer05a] 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.

[Fer05b] E.B.Fernandez and D. L. laRed M., "Using patterns to develop, evaluate, and teach secure operating systems", *Proceedings of the Congreso Internacional de Auditoría y Seguridad de la Información (**CIASI** 2005),* Madrid, Spain, 125-130.

[Fer06] E. B. Fernandez, T. Sorgente, and M. M. Larrondo-Petrie, "Even more patterns for secure operating systems," *Procs. of the Conference on Pattern Languages of Programs, PLoP 2006,* Portland, OR, October 2006, http://hillside.net/plop/2006/

[Fer08] E.B.Fernandez and D. laRed M., "Patterns for the secure and reliable execution of processes". *Procs. of the 15th Int.Conference on Pattern Languages of Programs (PLoP 2008),* Nashville, TN, Oct. 2008.

[Fer13] E.B.Fernandez, "*Security patterns in practice: Building secure architectures using software patterns*", Wiley Series on Software Design Patterns, 2013.

[For01] J.Forristal and G.Shipley, "Vulnerability assessment scanners", *Network Computing*, January 2001, 51-65, http://www.networkcomputing.com

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

[Gar96] S. Garfinkel and G. Spafford, *Practical Unix and Internet security* (2nd Edition), O'Reilly and Assocs., 1996.

[Gli84] V.D.Gligor, "A note on denial of service in operating systems", *IEEE Trans. on Software Eng.*, vol. SE-10, No 3, May 1984, 320-324.

[Gol84] B.Gold, R.R.Linde, and P.F.Cudney, "KVM/370 in retrospect", *Procs. of the 1984 Symposium on Security and Privacy*, 13-23, IEEE 1984.

[Gre04] T.C.Greene, "WinXP SP2 = Security placebo?", *The Register*, 09/02/04, http://www.theregister.co.uk/2004/09/02/winxpsp2_security_review

[Gus01] R. Guski et al., "Security on z/OS: Comprehensive,current, and flexible", *IBM Systems Journal*, vol. 40, No 3, 2001, 696-720.

[Ham73] K.J. Hammer Hodges, "A fault-tolerant multiprocessor design for real-time control", *Computer Design*, December 1973, 75-81.

[Har02] H. Hartig, "Security Architectures Revisited", *Proceedings of the 10th ACM SIGOPS European Workshop (EW 2002), September 22—25 2002, Saint-Emilion, France,* http://os.inf.tu-dresden.de/papers_ps/secarch.pdf

[HP] Hewlett Packard Corp., Virtual Vault, http://www.hp.com/security/products/virtualvault

[Hya] G. Hyatt, "Windows NT: How good is the security of Microsoft's newest operating system? *Comp. Sec. Journal*, Vol. 9, No 2.

[iee] IEEE, P2200 Base Operating System Security, http://bosswg.org

[Ivt] http://www.intel.com/technology/virtualization/technology.htm?iid=SEARCH

[Jai00] A. Jain, L. Hong, and S. Pankanti, "Biometric identification", *Comm. of the ACM*, vol. 43, No 2, February 2000, 91-98.

[Lan84] C.E. Landwehr and J.M.Carroll, "Hardware requirements for secure computer systems", *Procs. 1984 Symp. on Security and Privacy*, IEEE 1984, 34-40.

[Lev84] H.M. Levy, *Capability-based computer systems*, Digital Press, 1984.

[lin] Linux security, http://www.linuxsecurity.org

[Los00] P.A.Loscocco et al., "The inevitability of failure: The flawed assumption of security in modern computing environments", NSA, http://www.nsa.gov/selinux/inevit-abs.html

[Mat02] T. Matsumoto, "Importance of open discussion on adversarial analyses for mobile security technologies---A case study for user identification", http://www.itu.int/itudoc/itu-t/workshop/security/present/s5p4.pdf

[Mil94] B.Miller, "Vital signs of identity", *IEEE Spectrum*, February 1994, 22-30.

[Mil01] S.K.Miller, "The trusted OS makes a comeback", *Computer*, February 2001, 16-19.

[Mor79] R. Morris and K. Thompson, "Password security: A case history", *Comm. of the ACM*, vol. 22, No 11, November 1979, 594-597.

[Mud97] P. Mudge and Y. Benjamin, "Déjà vu all over again", *Byte*, November 1997, 81-86.

[nps] Naval Postgraduate School, C.I.S.R., VMM Project, http://cisr.nps.navy.mil/projectvmm.html

[NSA] National Security Agency, "Security-enhanced Linux", http://www.nsa.gov/selinux/index.html

[Pfl03] C.P.Pfleeger, *Security in computing*, 3rd Ed., Prentice-Hall, 2003.

[Rei87] B. Reid, "Reflections on some recent widespread computer break-ins", *Comm of the ACM*, vol. 30, No 2, February 1987, 103-105.

[Rin] http://en.wikipedia.org/wiki/Ring_(computer_security)

[Rod] J. Rodriguez and J. Klug, "Federated identity patterns in a service-oriented world", Microsoft Architecture Journal, 16, 6-11.

[Ros05] M. Rosenblum and T. Garfinkel, "Virtual machine monitors: Current technology and future trends", *Computer*, IEEE May 2005, 39-47.

[Rub94] C. Rubin and D. Arnovitz, "Virtual Vault white paper", Hewlett Packard, 1994-1999. http://www.hp.com/security/products/virtualvault/papers/whitepaper_3.1/

[Rus89] V. F. Russo and R. H. Campbell. ``Virtual Memory and Backing Storage Management in Multiprocessor Operating Systems Using Object-Oriented Design Techniques". *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, 267-278, October 1989.

[Sal75] J. H. Saltzer and M.D.Schroeder, "The protection of information in computer systems", *Procs. of the IEEE*, vol. 63, No 9, September 1975, 1278-1308. http://web.mit.edu/Saltzer/www/publications/protection/index.html

[Sha02] J.S.Shapiro and N. Hardy, "EROS: A principle-driven operating system from the ground up", *IEEE Software*, Jan./Feb. 2002, 26-33. See also: http://www.eros-os.org

[She02] K.M.Shelfer and J.D. Procaccino, "Smart card evolution", *Comm. of the ACM*, vol. 45, No 7, July 2002, 83-88.

[Sil12] A.Silverschatz, P.Galvin,and G.Gagne, *Operating System Concepts*, 9th Edition , J. Wiley, 2012.

[SOA] SOA Patterns, "Federated identity", http://soapatterns.org/federated_identity.php

[Son94] S. Sonntag et al., "Adaptability using reflection", *Procs. of the 27th Annual Hawaii Int. Conf. On System Sciences*, 1994, 363-392.

[Spe] R. Spencer, S. Smalley, P.Loscocco, M. Hibler, D. Andersen, and J. Lepreau, "The Flask security architecture: System support for diverse security policies", http://www.cs.utah.edu/flux/papers/micro/index.html

[Spo84] D.L. Spooner and E. Gudes, "A unifying approach to the design of a secure database operating system", *IEEE Trans. on Software Eng*., vol. SE-10, No 3, May 1984, 310-319.

[Sta99] W. Stallings, *Cryptography and network security: Principles and practice* (2nd Edition), Prentice-Hall, 1999.

[Sum97] R.C.Summers, *Secure Computing: Threats and Safeguards*, McGraw-Hill, 1997.

[Tan97] A.S. Tanenbaum and A.S. Woodhull, "Operating systems: Design and implementation", 2nd Ed., Prentice Hall, 1997.

[tru] "Linux Trustees (ACL) project", http://trustees.sourceforge.net

[Vie00] J.Viega and J.Voas, "The pros and cons of Unix and Windows security policies", *IT Professional*, Sept./October 2000, 40-45.

[vmw] Vmware, http://www.vmware.com

A verified kernel:
http://www.nicta.com.au/news/home_page_content_listing/world-first_research_breakthrough_promises_safety-critical_software_of_unprecedented_reliability