Chapter 2

The Security Architecture

Wherein the cryptlib security architecture is presented.

1. Security Features of the Architecture

Security-related functions which handle sensitive data pervade the architecture, which implies that security needs to be considered in every aspect of the design, and must be designed in from the start (it's very difficult to bolt on security afterwards). One standard reference [1] recommends that a security architecture have the properties listed below, with annotations explaining the approach towards meeting them as used in cryptlib:

- Permission-based access: The default access/use permissions should be deny-all, with access or usage rights being made selectively available as required. Objects are only visible to the process which created them, although the default object access setting makes it available to every thread in the process. The reason for this is because of the requirement for ease of use — having to explicitly hand an object off to another thread within the process would significantly reduce the ease of use of the architecture. For this reason the deny-all access is made configurable by the user, with the option of making an object available throughout the process or only to one thread when it is created. If the user specifies this behaviour when the object is created then only the creating thread can see the object unless it explicitly hands off control to another thread.
- Least privilege and isolation: Each object should operate with the least privileges possible to minimise damage due to inadvertent behaviour or malicious attack, and objects should be kept logically separate in order to reduce inadvertent or deliberate compromise of the information or capabilities they contain. These two requirements go hand in hand, since each object only has access to the minimum set of resources required to perform its task, and can only use those in a carefully controlled manner. For example if a certificate object has an encryption object attached to it, the encryption object can only be used in a manner consistent with the attributes set in the certificate object (it might be usable only for signature verification, but not for encryption or key exchange, or for the generation of a new key for the object).
- Complete mediation: Each object access is checked each time the object is used — it's not possible to access an object without this checking, since the act of mapping an object handle to the object itself is synonymous with performing the access check.
- Economy of mechanism and open design: The protection system design should be as simple as possible in order to allow it to be easily checked, tested, and trusted, and should not rely on security through obscurity. To meet this requirement, the security kernel is contained in a single module, which is divided into single-purpose functions of a dozen or so lines of code which were designed and implemented using "Design by Contract" principles [2], making the kernel very amenable to testing using mechanical verifiers such as ADL [3]. This is covered in more detail in later chapters.
- Easy to use: In order to promote its use, the protection system should be as easy to use and transparent as possible to the user. In almost all cases the user isn't even aware of the presence of the security functionality, since the programming interface can be set up to function in a manner which is almost indistinguishable from the conventional collection-of-functions interface.

A final requirement which is given is the separation of privilege, in which access to an object depends on more than one item such as a token and a password or encryption key. This is somewhat specific to user access to a computer system or objects on a computer system, and doesn't really apply to an encryption architecture.

The architecture employs a security kernel to implement its security mechanisms. This kernel provides the interface between the outside world and the architecture's objects (intra-object security) and between the objects themselves (interobject security). The security-related functions are contained in the security kernel for the following reasons [4]:

- Separation: By isolating the security mechanism from the rest of the implementation, it is easier to protect them from manipulation or penetration.
- Unity: All security functions are performed by a single code module.
- Modifiability: Changes to the security mechanism are easier to make and test.

- Compactness: Because it performs only security-related functions, the security kernel is likely to be small.
- Coverage: Every access to a protected object is checked by the kernel.

The details involved in meeting these requirements are covered in this and the following chapters.

1.1. Security Architecture Design Goals

Just as the software architecture is based on a number of design goals, so the security architecture, in particular the cryptlib security kernel, is also built on top of a number of specific principles. These are:

- Separation of policy and mechanism. The policy component deals with context-specific decisions about objects and requires detailed knowledge about the semantics of each object type. The mechanism deals with the implementation and execution of an algorithm to enforce the policy. The exact context and interpretation are supplied externally by the policy component. In particular it is important that the policy not be hard-coded into the enforcement mechanism, as is the case for a number of Orange Book-based systems. The advantage of this form of separation is that it then becomes possible to change the policy to suit individual applications (an example of which is given in the next chapter) without requiring the re-evaluation of the entire system.
- Verifiable design. It should be possible to apply formal verification techniques to the security-critical portion of the architecture (the security kernel) in order to provide a high degree of confidence that the security measures are implemented as intended (this is a standard Orange Book requirement for security kernels, although rarely achieved). Furthermore, it should be possible to perform this verification all the way down to the running code (this has never been achieved, for reasons covered in a later chapter).
- Flexible security policy. The fact that the Orange Book policy was hardcoded into the implementation has already been mentioned, a related problem was the fact that security policies and mechanisms were defined in terms of a fixed hierarchy which lead users who wanted somewhat more flexibility to try to apply the Orange Book as a Chinese menu in which they could choose one feature from column A and two from column B [5]. Since not all users require the same policy, it should be relatively easy to adapt policy details to user-specific requirements without either a great deal of effort on the part of the user or a need to re-evaluate the entire system whenever a minor policy change is made.
- Efficient implementation. A standard lament about security kernels built during the 1980's was that they provided abysmal performance. It should therefore be a primary design goal for the architecture that the kernel provide a high level of performance, to the extent that the user isn't even aware of the presence of the kernel.
- Simplicity. A simple design is required indirectly by the Orange Book in the guise of minimising the trusted computing base, however most kernels end up being relatively complex (although still simpler than mainstream OS kernels) because of the necessity to implementation a full range of operating system services. Because cryptlib doesn't require such an extensive range of services, it should be possible to implement an extremely simple, efficient, and easy-to-verify kernel design. In particular, the decision logic implementing the systems mandatory security policy should be encapsulated in the smallest and simplest possible number of system elements.

This chapter covers the security-relevant portions of the design, with later chapters covering implementation details and the manner in which the design and implementation are made verifiable.

2. Introduction to Security Mechanisms

The cryptlib security architecture is built on top of a number of standard security mechanisms which have evolved over the last three decades. This section contains an overview of some of the more common ones and the sections which follow discuss the details of how these security mechanisms are employed as well as detailing some of the more specialised mechanisms which are required for cryptlib's security.

2.1. Access Control

Access control mechanisms are usually viewed in terms of an access control matrix [6] which lists active subjects (typically users of a computer system) in the rows of the matrix and passive objects (typically files and other system resources) in the columns as shown in Figure 1. Because storing the entire matrix would consume far too much space once any realistic quantity of subjects or objects is present, real systems use either the rows or the columns of the matrix for access control decisions. Implementations which use a row-based implementation work by attaching a list of accessible objects to the subject, typically implemented using capabilities. Implementations which use a column-based implementation work by attaching a list of subjects allowed access to the object, typically implemented using access control lists (ACLs) or protection bits, a cut-down form of ACLs [7].

	Object1	Object2	Object3	
Subject1	Read/Write	Read	Execute	Capability
Subject2		Read	Execute	
Subject3	Read	Read		
		ACL		

Figure 1: Access control matrix

Capability-based systems issue capabilities or tickets to subjects which contain access rights such as read, write, or execute, and which the subject uses to demonstrate their right to access an object. Passwords are a somewhat crude form of capability which give up the fine-grained control provided by true capabilities in order to avoid requiring the user to remember and provide a different password for each object for which access is required. Capabilities have the property that they can be easily passed on to other subjects, and can limit the number of accessible objects to the minimum required to perform a specific task (for example a ticket could be issued which allowed a subject to access only the objects needed for the particular task at hand, but no more). The ease of transmission of capabilities can be an advantage but is also a disadvantage because the ability to pass them on can't be easily controlled, leading to a requirement that subjects maintain very careful control over any capabilities they possess and making revocation and access review (the ability to audit who has the ability to do what) extremely tricky.

ACL-based systems allow any subject to be allowed or disallowed access to a particular object. Just as passwords are a crude form of capabilities, so protection bits are a crude form of ACL's which are easier to implement but have the disadvantage that allowing or denying access to an object on a single-subject basis is difficult or impossible (for the most commonly-encountered implementation, Unix access control bits, single-subject control works only for the owner of the object, but not for arbitrary collections of subjects). Although groups of subjects have been proposed as a partial solution to this problem, the combinatorics of this solution make it rather unworkable, and they exhibit a single-group analog of the single-subject problem.

A variation of the access-control based view of security is the information-flow based view, which assigns security levels to objects and only allows information to flow to a destination object of an equal or higher security level than that of the source object [8]. This concept is the basis for the rules in the Orange Book, discussed in more detail below. In addition there exist a number of hybrid mechanisms which combine some of the best features of capabilities and ACLs or ones which try to work around the shortcomings of one of the two, for example by using the cached result of an ACL lookup as a capability [9], by providing per-object exception lists which allow capabilities to be revoked [10], or by extending the scope of one of the two approaches to incorporate portions of the other approach [11].

2.2. Reference Monitors

A reference monitor is the mechanism used to control access by a set of subjects to a set of objects as depicted in Figure 2. The monitor is the subsystem which is charged with checking the legitimacy of a subjects attempts to access objects, and represents the abstraction for the control over the relationships between subjects and objects. It should have the properties that it is tamper-proof, always invoked, and

simple enough to be open to a security analysis [12]. A reference monitor implements the "mechanism" part of the "separation of policy and mechanism" requirement.

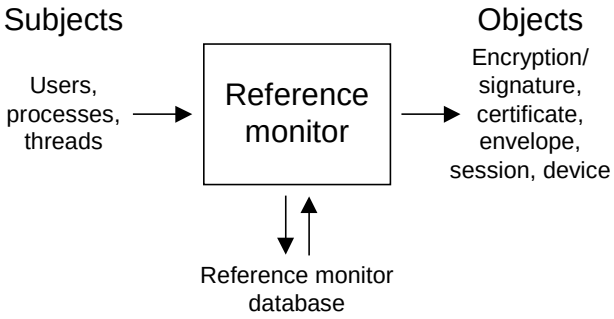

Figure 2: Reference monitor

2.3. Security Policies and Models

The security policy of a system is a statement of the restrictions on access to objects and/or information transfer which a reference monitor is intended to enforce, or more generally any formal statement of a system's confidentiality, availability, or integrity requirements. The security policy implements the "policy" part of the "separation of policy and mechanism" requirement.

The first widely-accepted formal security model, the Bell-LaPadula model [13], attempted to codify standard military security practices in terms of a formal computer security model. This model requires a reference monitor which enforces two security properties, the Simple Security Property and the *-Property (pronounced "star-property"[1][14]) using an access control matrix as the reference monitor database. The model assigns a fixed security level to each subject and object and only allows read access to an object if the subjects security level is greater than or equal to the objects security level (the simple security property, "no read up") and only allows write access to an object if the subjects security level is less than or equal to that of the objects security level (the *-property, "no write down"). The effect of the simple security property is to prevent a subject with a low-security level from reading an object with a high-security level (for example a user cleared for Secret data to read a Top Secret file). The effect of the *-property is to prevent a subject with a high security level from writing to an object with a low security level (for example a user writing Top Secret data to a file readable by someone cleared at Secret, which would allow the simple security property to be bypassed).

The intent of the Bell-LaPadula model beyond the obvious one of enforcing multilevel security (MLS) controls was to address the confinement problem [15], which required preventing the damage which could be caused by trojan horse software which could transmit sensitive information owned by a legitimate user to an unauthorised outsider. In the original threat model (which was based on multiuser mainframe systems) this involved mechanisms such as writing sensitive data to a location where the outsider could access it, in a commonly-encountered more recent threat model the same goal is achieved by using Outlook Express to send it over the Internet. Other, more obscure approaches were the use of timing or covert channels, in which an insider modules certain aspects of a systems performance such as its paging rate to communicate information to an outsider. The goals of the Bell-LaPadula model were formalised in the Orange Book (more formally the Department of Defense Trusted Computer System Evaluation Criteria or TCSEC [16]), which also added a number of other requirements and various levels of conformance and evaluation testing for implementations. A modification to the roles of the simple security and *- properties produced the Biba integrity model, in which a subject is allowed to write to an object of equal or lower integrity level and read from an object of equal or higher integrity level [17]. This model (although it reverses the way in which the two properties work) has the effect on integrity which the Bell-LaPadula version had on confidentiality. This type of mandatory integrity policy suffers from the problem that most system administrators have little familiarity

[1] When the model was initially being documented, no-one could think of a name so "*" was used as a placeholder to allow an editor to quickly find and replace any occurrences with whatever name was eventually chosen. No name was ever chosen, so the report was published with the "*" intact.

with its use, and there is little documented experience on applying it in practice (although what experience exists indicates that it, along with a number of other integrity policies, are awkward to manage) [18][19].

2.4. Security Models after Bell-LaPadula

After the Orange Book was introduced the so-called military security policy which it relied on was criticised as being unsuited for commercial applications which were often more concerned with integrity (the prevention of unauthorised data modification) than confidentiality (the prevention of unauthorised disclosure) — business equate trustworthiness with signing authority, not security clearances. One of the principal reactions to this was the Clark-Wilson model whose primary target was integrity rather than confidentiality. Instead of subjects and objects, this model works with constrained data items (CDIs) which are processed by two types of procedures, transformation procedures (TPs) and integrity verification procedures (IVPs). The TP transforms the set of a CDIs from one valid state to another, and the IVP checks that all CDI's conform to the system's integrity policy [20]. The Clark-Wilson model has close parallels in the transaction-processing concept of ACID properties [21][22][23], and is applied by using the IVP to enforce the precondition that a CDI is in a valid state and then using a TP to transition it, with the postcondition that the resulting state is also valid.

Another commercial policy which was targeted at integrity rather than confidentiality protection was Lipner's use of lattice-based controls to enforce the standard industry practice of separating production and development environments, with controlled promotion of programs from development to production and controls over the activities of systems programmers [24]. This type of policy was mostly just a formalisation of existing practice, although it was shown that it was possible to shoehorn the approach into a system which followed a standard MLS policy. Most other models were eventually subject to the same reinterpretation since during the 1980's and early 1990's it was a requirement that any new security model be shown to eventually map to Bell-LaPadula in some manner (usually via a lattice-based model, the ultimate expression of which was the Universal Lattice Machine or ULM [25]) in the same way that the US island-hopping campaign in WWII showed that you can get to Tokyo from anywhere in the Pacific if you were prepared to jump over enough islands on the way[2].

Another proposed commercial policy is the Chinese Wall security policy [26][27], which is derived from standard financial institution practice which is designed to ensure that objects owned by subjects with conflicting interests are never accessible by subjects from a conflicting interest group. In the real world this policy is used to prevent problems such as insider trading from occurring. The Chinese Wall policy groups objects into conflict-of-interest classes (that is, classes containing object groups for which there is a conflict of interest between the groups) and requires that a subject with access to a group of objects in a particular conflict-of-interest class can't access any other group of objects in that class, although they can access objects in a different conflict-of-interest class. Initially the subject has access to all objects in the conflict-of-interest class, but once they commit to one particular object access to any other object in the class is denied to them. In real-world terms, a market analyst might be allowed to work with Oil Company A (from the "Oil Company" conflict-of-interest class) and Bank B (from the "Bank" conflict-of-interest class), but not Oil Company B, since this would conflict with Oil Company A from the same class.

These basic models were intended to be used as general-purpose models and policies, applicable to all situations for which they were appropriate. Like other flexible objects such as rubber screwdrivers and foam rubber cricket bats, they give up some utility and practicality in exchange for their flexibility, and in practice tend to be extremely difficult to work with. The implementation problems associated in particular with the Bell-LaPadula/Orange Book model, with which implementers have the most experience, are covered in a later chapter, and newer efforts such as the Common Criteria (CC) have taken this flexibility-at-any-cost approach to a whole new level so that a vendor can do practically anything and still claim enough CC compliance to assuage the customer [28][3].

---

[2] Readers with too much spare time on their hands may want to try constructing a security model which requires two passes through (different views of) the lattice before arriving at Bell-LaPadula.

[3] One of the problems with the CC is that it's so vague — it even has a built-in metalanguage to help users try and describe what they're trying to achieve — that it's difficult to make any precise statement about it, which is why it isn't

Another problem which occurs with information-flow based models used to implement multilevel security policies is that information tends to flow up to the highest security level (a problem known as over-classification [29]), from which it is prevented from returning by the mandatory security policy. Examples of the type of problems this causes include users having to maintain multiple copies of the same data at different classification levels since once it's contaminated through access at level *m* it can't be moved back down to level *n*, the presence of inadvertent and annoying write-downs arising from the creation of temporary files and the like (multilevel secure Unix systems try to get around this with multiple virtual /tmp directories, but this doesn't really solve the problem for programs which attempt to write data to the user's home directory), problems with email where a user logged in at level *m* isn't even made aware of the presence of email at level *n* (when logged in at a low level a user can't see messages at high levels, and when logged in at a high level they can see messages at low levels but can't reply to them), and so on [30].

Although there have been some theoretical approaches made towards mitigating this problem [31], the standard solution is to resort to the use of trusted processes (pronounced "kludges"), technically a means of providing specialised policies outside the reach of kernel controls but in practice "a rug under which all problems not easily solved are swept" [32]. Examples of such trusted functions include an ability to violate the *-property in the SIGMA messaging system to allow users to downgrade over-classified messages without having to manually retype them at a lower classification level (leading to users leaking data down to lower classification levels because they didn't understand the policy being applied), the ability for the user to act as if they were simultaneously at multiple security levels under Multics in order to avoid having to log out at level *m* and in again at level *n* whenever they needed to effect a change in level, and the use of non-kernel security related (NKSR) functions in KSOS and downgrading functions in the Guard message filter to allow violation of the *-property so that functions such as printing could work [33]. cryptlib contains a single such mechanism, which is required in order to exchange session keys and to save keys held in encryption contexts (which are normally inaccessible) to persistent storage. This mechanism and an explanation of its security model is covered in a later section.

Alongside the general-purpose models outlined above and various other models derived from them [34][35][36], there are a number of application-specific models and adaptations which don't have the full generality of the previous models, but which in exchange offer a greatly reduced amount of implementation difficulty and complexity. Many of these adaptations came about because it was recognised that an attempt to create a one-size-fits-all model based on a particular doctrine (for example mandatory secrecy controls) didn't really work in practice, with the resulting system being both inflexible (hardcoding in a particular policy made it impossible to adapt the system to changing requirements) and unrealistic (it was very difficult to try to integrate diverse and often contradictory real-world policies to fit in with whatever policy was being used in the system at hand). As a result, more recent work has looked at creating blended security models or ones which incorporate more flexible, multi-policy mechanisms which allow the mixing and matching of features taken from a number of different models [37][38]. These multi-policy mechanisms might allow the mixing of mandatory and discretionary controls, Bell-LaPadula, Clark-Wilson, Chinese Wall, and other models, and a means of changing the policies to match changing real-world requirements when required. The cryptlib kernel implements a flexible policy of this nature through its kernel filter mechanisms which are explained in more detail in the next chapter.

The entire collection of hardware, firmware, and software protection mechanisms within a computer system which is responsible for enforcing security policy is known as the trusted computing base or TCB. In order to obtain the required degree of confidence in the security of the TCB, it needs to be made compact and simple enough for its security properties to be readily verified, which provides the motivation for the use of a security kernel as discussed in the next section.

2.5. Security Kernels and the Separation Kernel

Having covered security policies and mechanisms, we need to take a closer look at how the mechanism is to be implemented, and examine the most appropriate combination of policy and mechanism for our purposes.

---

mentioned in this work except to say that everything presented herein is bound to be compliant to some protection profile or other.

The practical expression of the abstract concept of the reference monitor is the security kernel, the motivation for whose use is the desire to isolate all security functionality, all critical components in a single place which can then be subject to analysis and verification. Since all non-kernel software is irrelevant to security, the immense task of verifying and securing an entire system is reduced to that of securing only the kernel [39]. The kernel provides the property that it "enforces security on the system as a whole without requiring the rest of the system to cooperate towards that end" [40].

The particular kernel type used in cryptlib is the separation kernel in which all objects are isolated from one another. This can be viewed as a variant of the noninterference requirement, which in its original form which was intended for use with MLS systems stipulated that high-level user input couldn't interfere with low-level user output [41] but which in this case requires that no input or output interfere with any other input or output.

The principles embodied in the separation kernel date back to the early 1960's with the concept of decomposable systems, ones where the components of the system have no direct interactions or only interact with similar components [42]. A decomposable system can be decomposed into two smaller systems with non-interacting components, which can in turn be recursively decomposed into smaller and smaller systems until they can't be decomposed any further. The separation kernel itself was first formalised in 1981 (possibly by more than one author [32]) with the realisation that secure systems could be modelled as a collection of individual distributed systems (in other words a completely decomposed system) in which security is achieved through the separation of the individual components, with mediation performed by a trusted component. The separation kernel allows such a virtually distributed system to be run within a single physical system, and provides the ability to compose a single secure system from individual modules which don't necessarily need to be as secure as the system as a whole [43][44]. Separation kernels are also known as separation machines or virtual machine monitors [45][46][47]. Following the practice already mentioned earlier, the separation kernel policy was also mapped to Bell-LaPadula in 1991 [45].

The fundamental axiom of the separation kernel's security policy is the isolation policy, in which a subject can only access objects which it owns. There is no inherent concept of information sharing or security levels, which greatly simplifies many implementation details. In Orange Book terms, the separation kernel implements a number of virtual machines equal to the number of subjects, running at system high. The separation kernel ensures that there is no communication between subjects by means of shared system objects (communications may, if necessary, be established using normal communications mechanisms, but not security-relevant functions). In this model each object is labelled with the identity of the subject which owns it (in the original work on the subject these identifying attributes were presented as colours) with the only check which needs to be applied to it being a comparison for equality rather than the complex ordering required in Bell-LaPadula and other models.

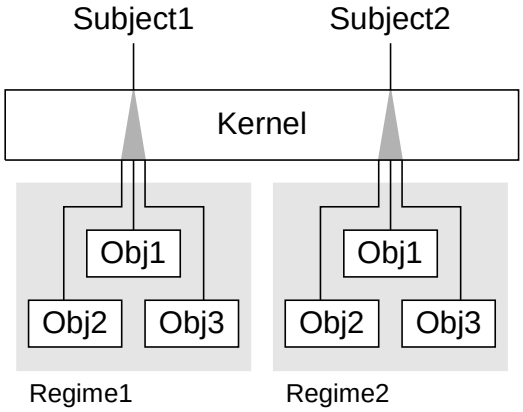

Figure 3: Separation kernel

An example of a separation kernel is shown in Figure 3, in which the kernel is controlling two groups of objects (referred to as regimes in the original work) owned by two different subjects. The effect of the separation kernel is that the two subjects can't distinguish the shared environment (the concrete machine)

from two physically separated ones with their resources dedicated to the subject (the abstract machines). The required security property for a separation kernel is that each regime's view of the concrete machine should correspond to the abstract machine, which leads to the concept of a proof of separability for separation kernels: If all communications channels between the components of a system are cut then the components of a system will become completely isolated from one another. In the original work, which assumed the use of shared objects for communication, this required a fair amount of analysis and an even longer formal proof [48], but the analysis in cryptlib's case is much simpler. Recall from the previous chapter that all inter-object communication is handled by the kernel, which uses its built-in routing capabilities to route messages to classes of objects and individual objects. In order to cut the communications channels, all we need to do is disable routing either to an entire object class (for example encryption contexts) or an individual object, which can be implemented through a trivial modification to the routing function. In this manner the complex data flow analysis required by the original method is reduced to a single modification (removing the appropriate routing information from the routing table used by the kernel routing code).

An early real-life implementation of the separation kernel concept is shown in Figure 4. This configuration connects multiple untrusted workstations through a LAN, with communications mediated by trusted network interface units (TNIUs) which perform the function of the separation kernel. In order to protect communications between TNIU's, all data sent over the LAN is encrypted and MAC'd by the TNIUs. The assumption made with this configuration is that the workstations are untrusted and potentially insecure, so that security is enforced by using the TNIUs to perform trusted mediation of all communication between the systems.

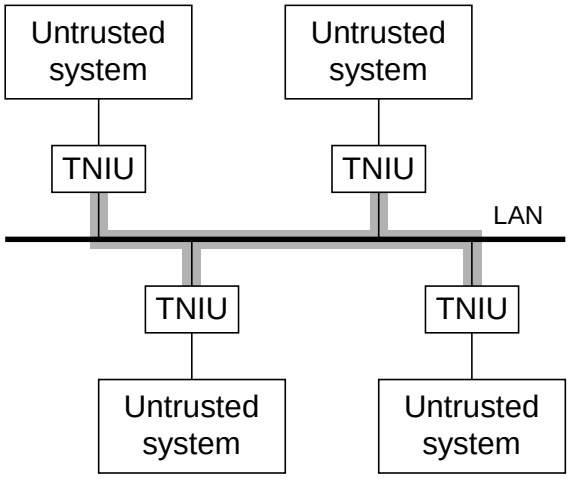

Figure 4: Separation kernel implemented using interconnected workstations

The advantage of a separation kernel is that complete isolation is much easier to attain and assure than the controlled sharing required by kernels based on models such as Bell-LaPadula, and that it provides a strong foundation upon which further application-specific security policies can be constructed. The reason for this, as pointed out in the work which introduced the separation kernel, is that "a lot of security problems just vanish and others are considerably simplified" [43]. Another advantage of the separation model over the Bell-LaPadula one is that it appears to provide a more rigorous security model with an accompanying formal proof of security [48], while some doubts have been raised over some of the assumptions made in, and the theoretical underpinnings of, the Bell-LaPadula model [49][50].

2.6. The Generalised TCB

The concept of the separation kernel has been extended into that of a generalised trusted computing base (GTCB), defined as a system structured as a collection of protection domains managed by a separation kernel [51]. In the most extreme form of the GTCB, separation can be enforced through dedicated hardware, typically by implementing the separation kernel using a dedicated processor. This is the approach used in the LOCK (LOgical Coprocessor Kernel, formerly known as the Secure Ada Target or SAT, before that the

Provably Secure Operating System or PSOS, and after LOCK the Secure Network Server or SNS, this project sheds its skin every few years in order to obtain funding for the "new" project) machine, which uses a special coprocessor (SIDEARM, which for performance reasons may consist of more than one physical CPU) which plugs into the system backplane to adjudicate access between the system CPU and memory [52][53]. Although originally used for performance reasons, this approach also provides a high level of security since all access control decisions are made by dedicated hardware which is inaccessible to any other code running on the system.

This type of implementation can be particularly appropriate in security-critical situations where the hardware in the host system isn't completely trusted. In practice this description applies (once a fine enough microscope is applied) to almost all systems, and is exacerbated by the fact that, while the software which comprises a trusted system is subject to varying levels of scrutiny, the hardware is generally treated as a black box, usually because there's no alternative available (the very few attempts to build formally verified hardware have only succeeded in demonstrating that this approach isn't really feasible [54][55]). Whereas in the 1970's and 1980's trusted systems, both hardware and software, were typically built by one company and could be evaluated as part of the overall system evaluation process, in the 1990's companies have moved to using commodity hardware, usually 80x86 architecture processors, while retaining the 1970's assumption that hardware was implicitly safe. As a result, anyone who can exploit one of the known security-relevant problem areas on a given CPU, or take advantage of a bug in a particular CPU family, or even discover a new flaw, could compromise an otherwise secure software design [56]. An example of this type of problem is the so-called unreal mode, in which a task running in real mode on an Intel CPU can address the entire 4GB address space even though it should only be able to see 1MB + 64KB (the extra 64KB is due to another slight anomaly in the way addressing is handled) [57]. Unreal mode has become so widely used after its initial discovery on the 80386 that Intel was forced to support it in all later processors, although its presence was never documented. Potential alternative avenues for exploits include the use of the undocumented ICEBP (in-circuit emulation breakpoint) instruction to drop the CPU into the little-documented ICE mode from which the system again looks like a 4GB DOS box, or the use of ICE mode or the somewhat less undocumented system management mode (SMM) to initialise the CPU into an otherwise illegal state, for example one which allows such oddities as a program running in virtual x86 mode in ring 0 [58]. This kind of trickery is possible because, when the CPU reloads the saved system state to move back into normal execution mode, it doesn't perform any checks on the saved state, allowing the loading of otherwise illegal values. Although no exploits using these types of tricks and other, similar ones are currently known, this is probably mostly due to their obscurity and the lack of demand for their use.

By moving the hardware which implements the kernel out of reach of any user code, the ability of malicious users to subvert the security of the system by taking advantage of particular features of the underlying hardware is eliminated, since no user can code ever run on the hardware which performs the security functions. With a kernel whose interaction with the outside world consists entirely of message passing (that is, one which doesn't have to manage system resources such as disks, memory pages, I/O devices, and other complications), such complete isolation of the security kernel is indeed possible.

2.7. Implementation Complexity Issues

When building a secure system for cryptographic use, there are two possible approaches which can be taken. The first is to build (or buy) a general-purpose kernel-based secure operating system and run the crypto code on top of it, and the second is to build a special-purpose kernel which is designed to provide security features which are appropriate specifically for cryptographic applications. Building the crypto code on top of an existing system is explicitly addressed by FIPS 140 [59], the one standard which specifically targets crypto modules. This requires that, where the crypto module is run on top of an operating system which is used to isolate the crypto code from other code, it be evaluated at progressively higher Orange Book levels for each FIPS 140 level, so that security level 2 would require the software module to be implemented on a C2-rated operating system. This provides something if an impedance mismatch between the actual security of equivalent hardware and software crypto module implementations, it's possible that the these security levels were set so low out of concern that setting them any higher would make it impossible to implement the higher FIPS 140 levels in software due to a lack of systems evaluated at that level. For example trying to source a B2 or more realistically B3 system to provide an adequate level of security for the crypto software is almost

impossible (the practicality of employing an OS in this class, whose members include Trusted Xenix, XTS 300, and Multos, speaks for itself).

Anther work which examines crypto software modules also recognises the need to protect the software through some form of security kernel based mechanism, but views implementation in terms of a device driver protected by an existing operating system kernel. The suggested approach is to modify an existing kernel to provide cryptographic support [60].

Two decades of experience in building high-assurance secure systems have conclusively shown that an approach which is based on the use of an application-specific rather than general-purpose kernel is the preferred one. For example in one survey of secure systems carried out during the initial burst of enthusiasm for the technology, most of the projects discussed were special-purpose filter or guard systems, and for the remaining general-purpose systems a recurring comment is of poor performance, occasional security problems, and frequent mentions of verification being left incomplete because it was too difficult (although this occurs for some of the special-purpose systems as well, and is covered in more detail in a later chapter) [61]. Although some implementors did struggle with the problem of kernel size and try to keep things as simple as possible (one paper predicted that "the KSOS, SCOMP, and KVM kernels will look enormous compared to our kernel" [62]), attempts to build general-purpose secure OS kernels appear to have foundered, leaving application-specific and special-purpose kernels as the best prospect for successful implementation.

One of the motivations for the original separation kernel design was the observation that other kernel design efforts at the time were targeted towards producing multilevel secure operating systems on general-purpose hardware, whereas many applications which required a secure system would be adequately served by a (much easier to implement) special-purpose, single-function system. One of the features of such a single-purpose system is that its requirements are usually very different from those of a general-purpose multilevel secure one. In real-world kernels, many processes require extra privileges in order to perform their work, which is impeded by the multilevel security controls enforced by the kernel. Examples of these extra processes include print spoolers, backup software, networking software, and assorted other programs and processes involved in the day-to-day running of the system. The result of this accumulation of extra processes is that the kernel is no longer the sole arbiter of security, so that all of the extra bits and pieces which have been added to the TCB now also have to be subject to the analysis and verification process. The need for these extra trusted processes has been characterised as "a mismatch between the idealisations of the multilevel security policy and the practical needs of a real user environment" [63].

An application-specific system, in contrast, has no need for any of the plethora of trusted hangers-on required by a more general-purpose system, since it performs only a single task which requires no further help from other programs or processes. An example of such a system is the NRL Pump, whose function is to move data between systems of different security levels under control of a human administrator, in effect transforming multiple single-level secure systems into a virtual multilevel secure system without the pain involved in actually building an MLS system. Communication with the pump is via non-security-critical wrappers on the high and low systems, and the sole function of the pump itself is that of a secure one-way communications channel which minimises any direct or indirect communications from the high system to the low system [64][65]. Because the pump performs only a single function, the complexity of building a full Orange Book kernel is avoided, leading to a much simpler and more practical design.

Another example of a special-purpose kernel is the one used in Blacker, a communications encryption device using an A1 application-specific kernel which in effect constitutes the entire operating system and acts as a mediator for interprocess communication [66]. At a time when other, general-purpose kernels were notable mostly for their lack of performance, the Blacker kernel performed at a level where its presence wasn't noticed by users when it was switched in and out of the circuit for testing purposes [67].

There is only one (known) system which uses a separation kernel in a cryptographic application, the NSA/Motorola Mathematically Analysed Separation Kernel (MASK) which is roughly contemporary with the cryptlib design and which is used in the Motorola Advanced Infosec Machine (AIM) [68][69]. The MASK kernel isolates data and threads (called strands) in separate cells, with each subject seeing only their own cell. In order to reduce the potential for subliminal channels, the kernel maintains very careful control over the use of resources such as CPU time (strands are non-preemptively multitasked, in effect making them fibers rather

than threads) and memory (a strand is allocated a fixed amount of memory which must be specified at compile time when it is activated), and has been carefully designed to avoid situations where a cell or strand can deplete kernel resources. Strands are activated in response to receiving messages from other strands, with message processing consisting of a one-way dispatch of an allocated segment to a destination under the control of the kernel [70]. The main concern for the use of MASK in AIM was its ability to establish separate cryptographic channels each with its own security level and cryptographic algorithm, although AIM also appears to implement a form of RPC mechanism between cells. Apart from the specification system used to build it [71], little else is known about the MASK design.

3. The cryptlib Security Kernel

The security kernel which implements the security functions outlined earlier is the basis of the entire architecture — all objects are accessed and controlled through it, and all object attributes are manipulated through it. The security kernel is implemented as an interface layer which sits on top of the objects, monitoring all accesses and handling all protection functions. The previous chapter presented the cryptlib kernel in terms of a message forwarding and routing mechanism which implements the distributed process software architectural model, but this only scratches the surface of its functionality: The kernel, whose general role is shown in Figure 5, is a full-scale Orange Book style security kernel which performs the security functions of the architecture as a whole.

As was mentioned earlier, the cryptlib kernel doesn't conform to the Bell-LaPadula paradigm because the types of objects which are present in the architecture don't correspond to the Bell-LaPadula notion of an object, namely a purely passive information repository. Instead, cryptlib objects combine both passive repositories and active agents represented by invocations of the object's methods. In this type of architecture information flow is represented by the flow of messages between objects, which are the sole source of both information and control flow [72].

The security kernel, the system element charged with enforcing the systemwide security policy, acts as a filter for this message flow, examining the contents of each message and allowing it to pass through to its destination (a forward information flow) or rejecting it as inappropriate and returning an error status to the sender. The replies to messages (a backward information flow) are subject to the same scrutiny, guaranteeing the enforcement of the security contract both from sender to recipient and from recipient back to the sender. The task of the kernel/message filter is to prevent illegal information flows, as well as enforcing certain other object access controls which are covered in a later section.

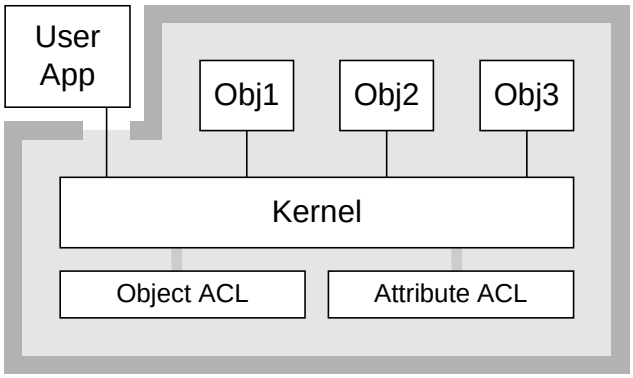

Figure 5. Architecture security model

The cryptlib kernel, serving as the reference monitor for a message-based architecture, has some similarities to the Trusted Mach kernel [73][74]. In both cases objects (in the Mach case these are actually tasks) communicate by passing messages via the kernel, however in the Mach kernel a task sends a message intended for another task to a message port for which the sender has send rights and the receiver has receive rights. The Mach kernel then checks the message and, if all is OK, moves it to the receiver's message queue

for which the receiver itself (rather than the kernel) is responsible for queue management. This system differs from the one used in cryptlib in that access control is based on send and receive rights to ports, leading to a number of complications as some message processing such as the queue management described above which might be better handled by the kernel is handed off to the tasks involved in the messaging. For example port rights may be transferred between the time the message is sent and the time it is received, or the port on which the message is queued may be deallocated before the message is processed, requiring extra interactions between the tasks and the kernel to resolve the problem. In addition the fact that Mach is a general-purpose operating system further complicates the message-passing semantics, since messages can be used to invoke other communications mechanisms such as kernel interface commands (KIC's) or can be used to arrange shared memory with a child process. In the cryptlib kernel, the only interobject communications mechanism is via the kernel, with no provision for alternate command and control channels or memory sharing.

A similar concept is used in the integrity-lock approach to database security, in which a trusted front-end (equivalent to the cryptib kernel) mediates access between an untrusted front-end (the user) and the database back-end (the cryptlib objects) [75][76], although the main goal of the integrity-lock approach is to allow security measures to be bolted onto an existing (relatively) insecure commercial database.

An alternative to having message-handling directly controlled by the kernel which has been suggested for use with object-oriented databases is for a special interface object to mediate the flow of messages to a group of objects. This scheme divides objects into protected groups and only allows communication within the group, with the exception of a single interface object which is allowed to communicate outside the group. Other objects, termed implementation objects, can only communicate within the group via the group's interface object. Inter-group communication is handled by making the interface object for one group an implementation object for a second group [77][78]. Figure 6 illustrates an example of such a scheme, with object 3 being an implementation object in group 1 and the interface object in group 2, and object 2 being the interface object for group 1.

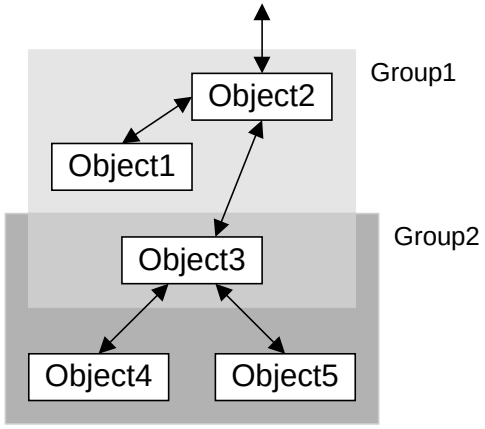

Figure 6: Access mediation via interface objects

Although this provides a means of implementing security measures where none would otherwise exist, it distributes enforcement of security policy across a potentially unbounded number of interface objects, each of which have to act as a mini-kernel to enforce security measures. In contrast the cryptlib approach of using a single, centralised kernel means that it's only necessary to get it right once, and allows a far more rigorous, controlled approach to security than the distributed approach involving mediation by interface objects.

A variant of this approach encapsulates objects inside a resource module (RM), an extended form of an object which controls protection, synchronisation, and resource access for network objects. The intent of a RM of this type, shown in Figure 7, is to provide a basic building block for network software systems [79]. As each message arrives, it is checked by the protection component to ensure that the type of access it is requesting is valid, has integrity checks (for example prevention of simultaneous access by multiple messages) enforced by the synchronisation component, and is finally processed by the access component.

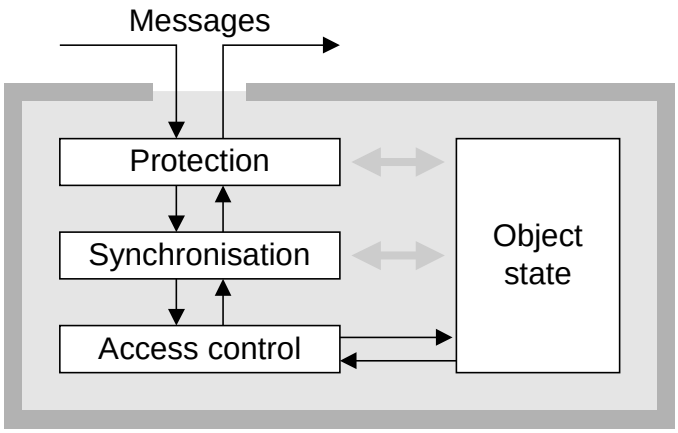

Figure 7: Object resource module

This approach goes even further than the use of interface objects since it makes each object/RM responsible for access control and integrity control/synchronisation. Again, with the cryptlib approach this functionality is present in the kernel which avoids the need to re-implement it (and get it right) for each individual object.

Another design feature which distinguishes the cryptlib kernel from many other kernels is that it doesn't provide any ability to run user code, which vastly simplifies its implementation and the verification process since there is no need to perform much of the complicated protection and isolation which is necessary in the presence of executable code supplied by the user. Since the user can still supply data which can affect the operation of the cryptlib code this doesn't do away with the need for all checking or security measures, but it does greatly simplify the overall implementation.

3.1. Extended Security Policies and Models

In addition to the basic message-filtering-based access control mechanism, the cryptlib kernel provides a number of other security services which can't be expressed using any of the security models presented so far. The most obvious shortcoming of the existing models is that none of them can manage the fact that some objects require a fixed ordering of accesses by subjects. For example an encryption context can't be used until a key and IV have been loaded, but none of the existing security models provides a means for expressing this requirement. In order to constrain the manner in which subjects can use an object, we require a means of specifying a sequence of operations which can be performed with the object, a mechanism first introduced in the form of transaction control expressions which can be used to enforce serialisability of operations on and with an object [80][81]. Although the original transaction control expression model required the additional property of atomicity of operation (so that either none or all of the operations of a transaction could take effect), this property isn't appropriate for the operations performed by cryptlib and isn't used. Another approach which can be used to enforce serialisation is to incorporate simple boolean expressions into the access control model to allow the requirement for certain access sequences to be expressed [82][83] or even to build sequencing controls using finite state automata encoded in state transition tables [84][85], but again these aren't really needed in cryptlib.

Since cryptlib objects don't provide the capability for performing arbitrary operations, cryptlib can use a greatly simplified form of serialisability control which is tied into the object life cycle described in the next section. This takes advantage of the fact that an object is transitioned through a number of discrete states by the kernel during its life cycle so that only operations appropriate to that state can be allowed. For example when an encryption context is in the "no key loaded" state encryption is disallowed but a key load is possible, while an object in the "key loaded" state can be used for encryption but can't have a new key loaded over the top of the existing one. The same serialisability controls are used for other objects, for example a certificate can have its attributes modified before it is signed but not after it is signed.

Another concept which is related to transaction control expressions is that of transaction authorisation controls, which were designed to manage the transactions which a user can perform against a database. An

example of this type of control is that a user may be authorised to run the "disburse payroll" transaction, but wouldn't be authorised to perform an individual payroll disbursement [86]. cryptlib includes a similar form of mechanism which is applied when lower-layer objects are controlled by higher-layer ones, for example a user might be permitted to process data through an envelope container object, but wouldn't be permitted to directly access the encryption, hashing, or signature contexts which the envelope is using to perform its task. This type of control is implicit in the way the higher-level objects work, and doesn't require any explicit mechanism support within cryptlib beside the standard security controls.

With the benefit of 20/20 hindsight coming from other researchers who have spent years exploring the pitfalls which inevitably accompany any security mechanism, cryptlib takes precautions to close certain security holes which can crop up in existing designs. One of these is the general inability of ACL-based systems to constrain the use of the privilege to grant privileges, which gives capability fans something to respond with when ACL fans criticise capability-based systems on the basis that they have problems tracking which subjects have access to a given object (that is, who holds a capability). One approach to this problem has been to subdivide ACLs into two classes, regular and restricted, and to greatly constrain the modification of restricted ACLs in order to provide greater control over the distribution of access privileges, and to provide limited privilege transfer in which the access rights which are passed to another subject are only temporary [87]. Another approach is that of owner-retained access control (ORAC) or propagated access control (PAC), which gives the owner of an object full control over it and allows them to later add or revoke privileges which propagate through to any other subjects who have access to it, effectively making the controls discretionary for the owner and mandatory for everyone else [88][89]. This type of control is targeted specifically for intelligence use, in particular NOFORN and ORCON-type controls on dissemination, and would seem to have little other practical application, since it both requires the owner to act as the ultimate authority on access control decisions and gives them (or a trojan horse acting for them) the ability to allow anyone full access to an object.

cryptlib objects face a more general form of this problem because of their active nature, since not only access to but also use of the object needs to be controlled. For example although there is nothing much to be gained from anyone reading the key size attribute of a private-key object (particularly since the same information is available through the public key), it is extremely undesirable for anyone to be able to repeatedly use it to generate signatures on arbitrary data. In this case "anyone" also includes the key owner, or at least trojan horse code acting as the owner.

In order to provide a means of controlling these problem areas, the cryptlib kernel provides a number of extra ACLs which can't be easily expressed using any existing security model. These ACLs can be used to restrict the number of times an object can be used (for example a signature object might be usable to generate a single signature after which any further signature operations would be disallowed), restrict the types of operations an object can perform (for example an encryption context representing a conventional encryption algorithm might be restricted to allowing only encryption or only decryption of data), provide a dead-man timer to disable the object after a given amount of time (for example a private-key object might disable itself five minutes after it has been created to protect against problems when the user is called away from their computer after activating the object but before they can use it), and a number of other special-case circumstances. These object usage controls are rather specific to the cryptlib architecture and are relatively simple to implement since they don't require the full generality or flexibility of controls which might be needed for a general-purpose system.

3.2. Controls Enforced by the Kernel

As the previous sections have illustrated, the cryptlib kernel enforces a number of controls adapted from a variety of security policies, as well as introducing new application-specific ones which apply specifically to the cryptlib architecture. Table 1 summaries the various types of controls and their implications and benefits, alongside some more specialised controls which are covered in later sections.

Policy	Separation
Section	2.5. Security Kernels and the Separation Kernel
Type	Mandatory
Description	All objects are isolated from one another and can only communicate via the kernel
Benefit	Simplified implementation and the ability to use a special-purpose kernel which is very amenable to verification.
Policy	No ability to run user code
Section	3. The cryptlib Security Kernel
Type	Mandatory
Description	cryptlib is a special-purpose architecture with no need for the ability to run user-supplied code. Users can supply data to be acted upon by objects within the architecture, but can't supply executable code
Benefit	Vastly simplified implementation and verification
Policy	Single-level object security
Section	3. The cryptlib Security Kernel
Type	Mandatory
Description	There is no information sharing between subjects so there is no need to implement an MLS system. All objects owned by a subject are at the same security level, although object attributes and usages are effectively multilevel.
Benefit	Simplified implementation and verification
Policy	Multilevel object attribute and object usage security
Section	6. Object Usage Control
Type	Mandatory
Description	Objects have individual ACLs indicating how they respond to messages which affect attributes or control the use of the object from subjects or other objects.
Benefit	Separate controls are allowed for messages coming from subjects inside and outside the architectures security perimeter, so that any potentially risky operations on objects can be denied to subjects outside the perimeter.
Policy	Serialisation of operations with objects
Section	3.1 Extended Security Policies and Models, 4. The Object Life Cycle
Type	Mandatory
Description	The kernel controls the order in which messages may be sent to objects, ensuring that certain operations are performed in the correct sequence.
Benefit	Kernel-mandated control over how objects are used, removing the need for explicit checking in each objects implementation.
Policy	Object usage controls
Section	3.1 Extended Security Policies and Models
Type	Mandatory/discretionary
Description	Extended control over various types of usage such as whether an object can be used for a particular purpose and how many times an object can be used before access is disabled.
Benefit	Precise user control over the object so that, for example, a signing key can only be used to generate a single signature under the direct control of the user rather than an uncontrolled number of signatures under the control of a trojan horse.

Table 1: Controls and policies enforced by the cryptlib kernel

4. The Object Life Cycle

Each object goes through a series of distinct stages during its lifetime. Initially the object is created in the uninitialised state by the kernel, after which it hands it off to the object-type-specific initialisation routines to perform object-specific initialisation and set any attributes which are supplied at object creation (for example the encryption algorithm for an encryption context or the certificate type for a certificate object). The attributes which are set at object creation time can't be changed later. Once the kernel and object-specific initialisation is complete the object is in the low state, in which object attributes can be read, written, or deleted, but the object can't generally be used for its intended purpose. For example in this state a conventional encryption context can have its encryption mode and IV set, but it can't be used to encrypt data because no key is loaded.

At some point the object receives a trigger message which causes the kernel to move it into the high state in which access to attributes is greatly restricted but the object can be used for its intended purpose. For the aforementioned context the trigger message would be one which loads or generates a key in the context, after which encryption with the context becomes possible but operations such as loading a new key over the top of the existing one are disallowed. The object life cycle is shown in Figure 8. As indicated by the arrows, the progression through these stages is strictly one-way, with the kernel ensuring that, like military security levels, the object progresses to higher and higher levels of security until it is eventually destroyed by the kernel at the end of its life (the pre- and post-use states which exist outside the normal concept of object states have also been described as alpha and omega states, being respectively C++ objects before their constructor is called and after their destructor is called, "the object declaration before it is constructed and the carcass of an object after it has been deleted" [90]).

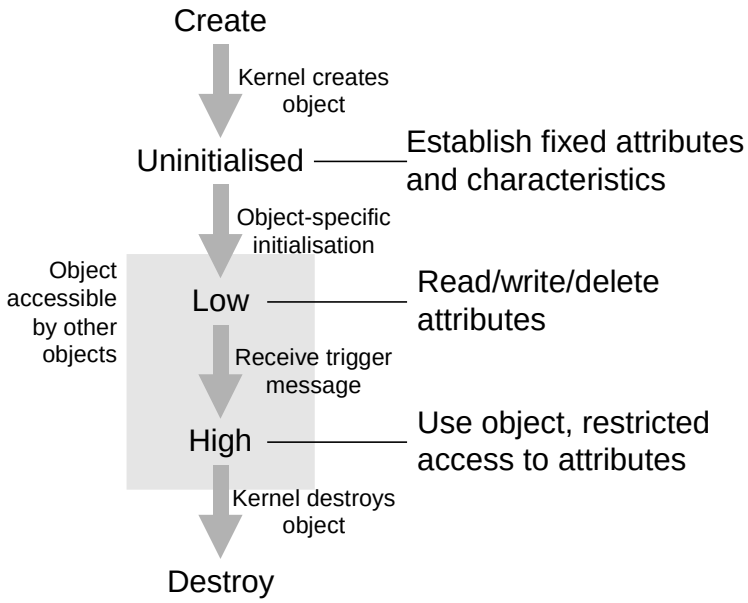

Figure 8: The Object Life Cycle

Similar life cycles occur with other objects, for example a certificate is transitioned into the high state when it is signed by a CA key and a session object is transitioned into the high state once a connection with a peer is established.

Although the cryptlib architecture doesn't restrict the number of states to only two (low and high), in practice only these two are used in order to avoid the combinatorial explosion of states which would occur if every change in an object's internal state were to be mapped to a cryptlib state. Even something as simple as an encryption context could have states corresponding to various combinations of encryption mode set or not set, IV set or not set, and key loaded or not loaded, and the number of states attainable by more complex object types such as envelope or session container objects doesn't bear thinking about. For this reason objects are

restricted to having only two states, experience with cryptlib has shown that this is adequate for handling all eventualities.

4.1. Object Creation and Destruction

When an object is created, it is identified to the entity which requested its creation through an arbitrary handle, an integer value which has no connection to the object's data or associated code. The handle represents an entry in an internal object table which contains information such as a pointer to the objects data and ACL information for the object. The handles into the table are allocated in a pseudorandom manner not so much for security purposes but to avoid the problem of the user freeing a handle by destroying an object and then immediately having the handle reused for the next object allocated, leading to problems if some of the user's code still expects to find the previous object accessible through the handle. If the object table is full, it is expanded to make room for more entries.

Both the object table and the object data are protected through locking and ACL mechanisms. Creation of a new object proceeds as shown in Figure 9, which creates an object of the given type with the given attributes, adds an entry for it to the object table, marks it as under construction so that it can't be accessed in the incomplete state, and returns a pointer to the object data to the caller (the caller being code within cryptlib itself, the user never has access to this level of functionality). The object can also have a variety of attributes specified for its creation such as the type of memory used, for example some systems can allocate limited amounts of protected, non-pageable memory which is preferred for sensitive data such as encryption contexts.

The object is now in the uninitialised state. At this point the caller can complete any object-specific initialisation, after which it sends an "init complete" message to the kernel which sets the object's state to normal, unlocks the object and returns its handle to the user. At this point the object is in the low state ready for use. When the object was initially created by the kernel, it set an ACL entry which marked it as being visible only within the architecture, so that the calling routine has to explicitly make it accessible outside the architecture by changing the ACL (that is, it defaults to deny-all rather than permit-all).

```
caller requests object creation by kernel

lock object table;
create new object with requested type and attributes;
if( object was created successfully )
   add object to object table;
   set object state = under construction;
unlock object table;

caller completes object-specific initialisation
caller sends initialisation complete message to kernel

lock object table;
set object state = normal;
unlock object table;
```

Figure 9: Object creation

An object is usually destroyed by having its reference count decremented sufficiently that it drops to zero, which causes the kernel to destroy the object. Before the object itself is destroyed, any dependent objects (for example a public-key context attached to a certificate) also have their reference counts decremented, with the process continuing recursively until leaf objects are reached. Destruction of an object proceeds as shown in Figure 10, which signals any dependent objects which may be present, marks the object as being in the process of being destroyed so that it can't be accessed any more while this is in progress, sends a destroy object message to the object's message handler to allow it to perform object-specific cleanup, and finally removes the object from the kernel object table.

```
caller requests decrement of object's reference count

lock object table;
decrement reference count of any dependent objects;
set object state = being destroyed;
unlock object table;

send destroy object message to object's message handler
```

```
lock object table;
dequeue any further messages for this object;
clear entry in object table;
unlock object table;
```

Figure 10: Object destruction

At this point the object's slot in the object table is ready for reuse. As has been mentioned previously, in order to avoid problems where a newly-created object would be entered into a newly-freed slot and be allocated the same handle as the previous object in that slot, leading to a potential for confusion if the users code isn't fully aware that such a replacement has taken place and that the handle now belongs to a new object, the kernel cycles through the slots to ensure that handles aren't reused for the longest time possible in the same way that Unix process IDs are cycled to give the longest possible time before ID reuse. An alternative approach, used in the LOCK kernel, is to encrypt the unique IDs (UIDs) which it uses, although this is motivated mainly by a need to eliminate the potential for covert channel signalling via UIDs (which isn't an issue with cryptlib) and by the ready availability of fast crypto hardware, which is an integral portion of the LOCK system.

Note that for both object creation and object destruction the object-specific processing is performed with the object table unlocked, which ensures that if the object-specific processing takes a long time to complete or even hangs altogether, the functioning of the kernel isn't affected.

5. Object Access Control

Each object within the architecture is contained entirely within its security perimeter, so that data and control information can only flow in and out in a very tightly-controlled manner, and that objects are isolated from each other within the perimeter by the security kernel. Associated with each object is a mandatory access control list (ACL) which determines who can access a particular object and under which conditions the access is allowed. Mandatory ACLs control features such as whether an object can be accessed externally (by the user) or only via other objects within the architecture (for example an encryption context associated with an envelope can only be used to encrypt data by the envelope, not by the user who owns the envelope), the way in which an object can be used (for example a private key may be usable for decryption but not for signing), and so on.

A somewhat special-case ACL entry is the one which is used to determine which processes or threads can access an object. This entry is set by the object's owner either when it is created or at a later point when the security properties of the object are changed, and provides a much finer level of control than the internal/external access ACL. Since an object can be bound to a process or a thread within a process by an ACL, it will be invisible to other processes or threads, resulting in an access error if an attempt is made to access it from another process or thread.

A typical example of this ACL's use is shown in Figure 11, which illustrates the case of an object created by a central server thread setting up a key in the object and then handing it off to a worker thread which uses it to encrypt or decrypt data. This model is typical of multithreaded server processes which use a core server thread to manage initial connections and then hand further communications functions off to a collection of active threads.

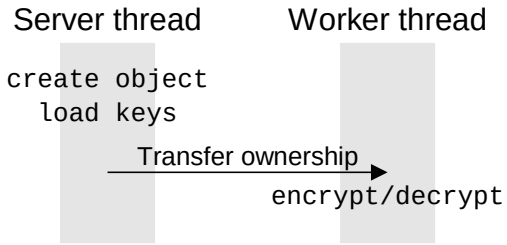

Figure 11. Object ownership transfer

Operating at a much finer level of control than the object ACL is the discretionary access control (DACL) mechanism through which only certain capabilities in an object may be enabled. For example once an encryption context is established, it can be restricted to only allow basic data encryption and decryption, but not encrypted session key export. In this way a trusted server thread can hand the context off to a worker thread without having to worry about the worker thread exporting the session key contained within it[4]. Similarly, a signature object can have a DACL set which allows it to perform only a single signature operation before it is automatically disabled by the security kernel, closing a rather troublesome security hole in which a crypto device such as a smart card can be used to authenticate arbitrary numbers of transactions by a rogue application.

These ACLs aren't true DACLs in the sense that they can't be arbitrarily changed by the owner once set. Some of the DACLs are one-shot so that once set they can't be unset, and others can be altered initially but can then be locked down using a one-shot ACL at which point they can no longer be changed. For example a subject can set properties such as controls on object usage as required and then lock them down so that no further changes can be made. This might be done when a keyset or device is used to instantiate a certificate object and wishes to place controls on the way it can be used before making the object accessible to the user. Since the ACLs are now mandatory, they can't be reset by the user.

ACLs are inherited across objects, so that retrieving a private key encryption object from a keyset container object will copy the container object's ACL across to the private key encryption object.

5.1. Object Security Implementation

The actions performed when the user passes an object's handle to cryptlib are shown in Figure 12. This performs the necessary ACL checking for the object in an object-independent manner. The link from external handles through the kernel object table and ACL check to the object itself is shown in Figure 13.

```
lock object table;
verify that the handle is valid;
verify that the ACL allows this type of access;
if( access allowed )
   set object state = processing message;
   further messages will be enqueued for later processing
   unlock object table;
   forward message to object;
   lock object table;
   set object state = normal;
unlock object table;
```

Figure 12: Object access during message processing

The kernel begins by performing a number of general checks such as whether the message target is a valid object, whether the message is appropriate fro this object type, whether this type of access is allowed, and a variety of other checks of which more details are given in later sections and in the next chapter. It then sets a flag in the object table to indicate that the object is busy processing a message so that further messages which arrive will be enqueued, unlocks the object table so that other messages may be processed, and forwards the message to the object. When the object has finished processing the message the kernel resets its state so that further messages may be processed by it. Again, there are a range of security controls applied during this process which are described later.

[4] Obviously chosen-plaintext and similar attacks are still possible, but this is something which can never be fully prevented, and which provides an attacker far less opportunity than the presence of a straight key export facility.

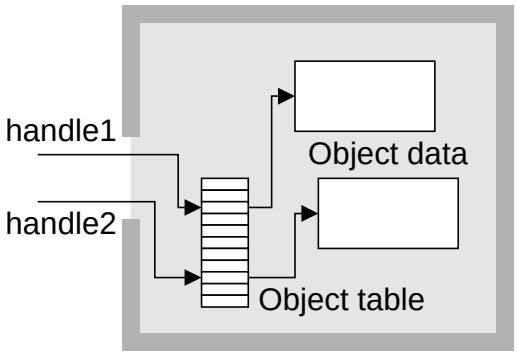

Figure 13. Object ACL checking

The access check is performed each time an object is used, and the ACL used is attached to the object itself rather than to the handle. This means that if an ACL is changed, the change immediately affects all users of the object rather than just the owner of the handle which changed the ACL. This is in contrast to the Unix security model in which an access check is performed once when an object is instantiated (for example when a file is created or opened) and the access rights which were present at that time remain valid for the lifetime of the handle to the object. For example if a file is temporarily made world-readable and a user opens it, the handle remains valid for read access even if read permission to the file is subsequently removed — the security setting applies to the handle rather than to the object and can't be changed after the handle is created. cryptlib instead applies its security to the object itself, so that a change in an objects ACL is immediately reflected to all users of the object. Consider the example in Figure 14, in which an envelope contains an encryption context accessed either through the internal handle from the envelope or the external handle from the user. If the user changes the ACL for the encryption context the change is immediately reflected on all users of the context, so that any future use of the context by the envelope will result in access restrictions being enforced using the new ACL.

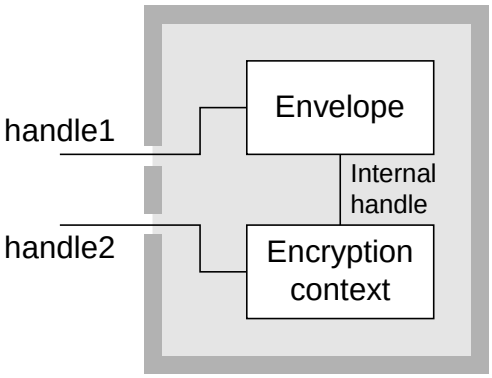

Figure 14. Objects with multiple references

Each object can be accessible to multiple threads or to a single thread. The thread access ACL is handled as part of the thread locking mechanism which is used to make the architecture thread-safe, and tracks the identity of the thread which owns the object. By setting the thread access ACL, a thread can claim an un-owned object, relinquish a claim to an owned object, and transfer ownership of an object to another thread. In general critical objects such as encryption contexts will be claimed by the thread which created them and will never be relinquished until the object is destroyed — to all other threads in the process the object doesn't appear to exist.

5.2. External and Internal Object Access

cryptlib distinguishes between two types of object access, accesses from within the cryptlib security perimeter and accesses from outside the perimeter. When an object is created, its ACLs are set so that it is only visible from within the security perimeter so that even if code outside the perimeter guesses the object's handle and tries to send a message to it, the message will be blocked by the kernel which will report that the object doesn't exist (as far as the outside user is concerned it doesn't, since it can't be accessed in any way).

Objects which are used by other objects (for example a public or private key context attached to a certificate, or a hash or encryption context attached to an envelope or session object) are left in this state and can never be directly manipulated by the user, but can only be used in the carefully-controlled manner permitted by the object which owns them. This is usually done by having the owning object set appropriate ACLs for the dependent object and letting the kernel enforce access controls rather than having the owning object act as an arbitrator, although in the case of very fine-grained and not particularly security-critical controls which the kernel doesn't manage, examples being the certificate expiry date and extended certificate usage such as emailProtection, the control would be handled by the owning object. Objects created directly by the user on the other hand have their ACLs set to allow access from the outside. In addition objects created by internal objects but destined for use by the user (for example public-key or certificate objects instantiated via a keyset object) also have their ACLs set to allow external access.

In addition to distinguishing between internal and external accesses to objects as a whole, cryptlib also applies this distinction to object attributes and usage. Although the objects themselves inherently exist at a single security level since there's no way for cryptlib to control data sharing among subjects so there's little need to provide a MLS mechanism for objects, the actual attributes and usage modes for the object have two distinct sets of ACLs, one for attribute manipulation and usage messages coming from the outside (the user) and one for messages coming from the inside (other objects).

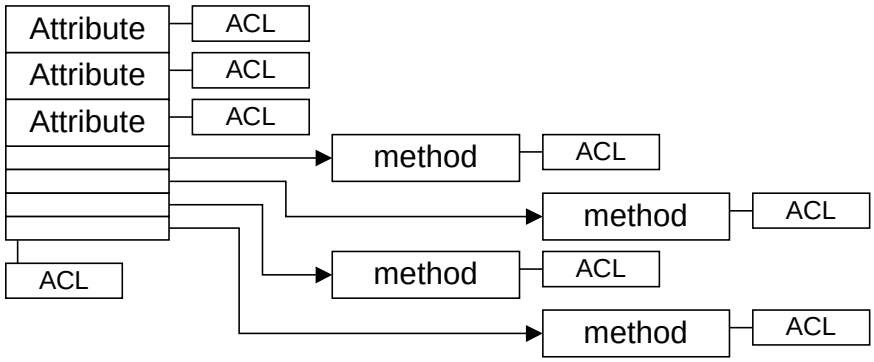

Figure 15: Annotated object internal structure indicating presence of ACLs

The object ACLs can best be visualised by annotating the object internal structure diagram from the previous chapter in the manner shown in Figure 15, which illustrates that not only the object as a whole but each individual attribute and method (in other words the way in which the object can be used) have their own individual ACLs. These controls, although conceptually a part of the object, are maintained and enforced entirely by the kernel, and are discussed in the following sections.

6. Object Usage Control

As Figure 15 indicates, every action which can be performed by an action object has its own ACL associated with it. The ACL defines the conditions under which the action can be performed, or more precisely the permission which the sender must have in order for the kernel to allow the corresponding message through to the object.

The default setting for an ACL is ACTION_PERM_NOTAVAIL, which indicates that this action isn't available for the object (for example an encrypt action message for a hash context would have an ACL set to

ACTION_PERM_NOTAVAIL). This setting corresponds to an all-zero value for the ACL, so that the default initialised-to-zero value constitutes a deny-all ACL.

The next level is ACTION_PERM_NONE, which means that the action is in theory available but has been disallowed. An example of this is a private key which could normally be used for signing and decryption but which has been constrained to only allow signing, so that the decrypt action message will have an ACL setting of ACTION_PERM_NONE.

The final two levels enable the action message to be passed on to the object. The more restrictive setting is ACTION_PERM_NONE_EXTERNAL, which means that the action is only permitted if the message originates from another object within the cryptlib security perimeter. If the message comes from the outside (in other words from the user), the result is the same as if the ACL were set to ACTION_PERM_NONE. This ACL setting is used in cases such as signature envelopes, where the envelope can send a sign data message to an attached private key context to sign the data it contains but the user can't directly send the same message to the context.

The least restrictive permission is ACTION_PERM_ALL, which means the action is available for anyone.

The cryptlib kernel enforces a ratchet for these setting which implements the equivalent of the *-property for permissions in that it only allows them to be set to a more restrictive value than their existing one. This means that if an ACL for a particular action is currently set to ACTION_PERM_NONE_EXTERNAL, it can only be changed to ACTION_PERM_NONE, but never to ACTION_PERM_ALL. Once set to ACTION_PERM_NONE, it can never be returned to its original (less restrictive setting).

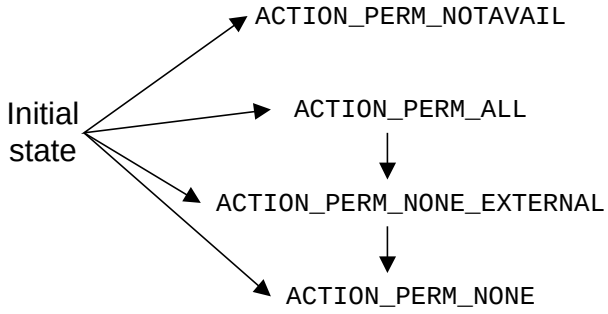

Figure 16: State machine for object action permissions

The finite state machine in Figure 16 indicates the transitions which are allowed by the cryptlib kernel. On object creation the ACLs may be set to any level, but after this the kernel-enforced *-property applies and the ACL can only be set to a more restrictive setting.

6.1. Permission Inheritance

The previous chapter introduced the concept of dependent objects in which one object, for example a public-key encryption context, was tied to another, in this case a certificate. The certificate usually specifies among various other things constraints on the manner in which the key can be used, for example it might only allow use for encryption or for signing or key agreement. In a conventional implementation an explicit check for which types of usage are allowed by the certificate needs to be made before each use of the key. If the programmer forgets to make the check, or gets it wrong, or never even considers the necessity of such a check (there are implementations which do all of these), the certificate is useless because it doesn't provide any guarantees about the manner in which the key is used.

The fact that cryptlib provides ACLs for all messages sent to objects means that we can remove the need for programmers to explicitly check whether the requested access or usage might be constrained in some way since the kernel can perform the check automatically as part of its reference monitor functionality. In order to do this, we need to modify the ACL for an object when another object is associated with it, a process which is again performed by the kernel. This is done by having the kernel check which way the certificate constrains the use of the context and adjusting the context's access ACL as appropriate. For example if the certificate

responded to a query of its signature capabilities with a permission denied error than the context's signature action ACL would be set to ACTION_PERM_NONE. From then on, any attempt to use the context to generate a signature would be automatically blocked by the kernel.

There is one special-case situation which occurs when a context is attached to a certificate for the first time when a new certificate is being created. In this case the context's access ACL isn't updated for that one instantiation of the object because the certificate may constrain the context in a manner which makes its use impossible. Examples of instances where this can occur are when creating a self-signed encryption-only certificate (the kernel would disallow the self-signing operation) or when multiple mutually exclusive certificates are associated with a single key (the kernel would disallow any kind of usage). The semantics of both of these situations are in fact undefined, falling into one of the many black holes which X.509 leaves for implementors (self-signed certificates are generally assumed to be version 1 certificates which don't constrain key usage, and the fact that people would issue multiple certificates for a single key wasn't really considered). As a following chapter will illustrate, the fact that cryptlib implements a formal, consistent security model reveals these problems in a manner which a typical ad hoc design would never be able to do. Unfortunately in this case the fact that the real world isn't consistent or rigorously defined means that it's necessary to provide this workaround to meet user's expectations. In cases where users are aware of these constraints, the exception can be removed and cryptlib can implement a completely consistent policy with regard to ACLs.

One additional security consideration needs to be taken into account when the ACLs are being updated. Because a key with a certificate attached indicates that it's (probably) being used for some function which involves interaction with a relying party, the access permission for allowed actions is set to ACTION_PERM_NONE_EXTERNAL rather than ACTION_PERM_ALL to both ensure that the object is only used in a safe manner via cryptlib internal mechanisms such as enveloping, and to make sure that it's not possible to utilise the signature/encryption duality of public-key algorithms like RSA to create a signature where it's been disallowed by the ACL. This means that if a certificate constrains a key to being usable for encryption only or for signing only, the architecture really will only allow its use for this purpose and no other. Contrast this with approaches like PKCS #11, where controls on object usage are trivially bypassed through assorted creative uses of signature and encryption mechanisms, and in some cases even appear to be standard programming practice.

6.2. The Security Controls as an Expert System

The object usage controls represent an extremely powerful means of regulating the manner in which an object can be used. Their effectiveness is illustrated by the fact that they caught an error in smart cards issued by a European government organisation which incorrectly marked a signature key stored on the cards as a decryption key. Since the accompanying certificate identified it as a signature-only key, the union of the two was a null ACL which didn't allow the key to be used for anything. This error had gone unnoticed by other implementations.

The complete system of ACLs and kernel-based controls in fact extends beyond basic error-checking applications to form an expert system which can be used to answer queries about the properties of objects. Loading the knowledge base involves instantiating cryptlib objects from stored data such as certificates or keys, and querying the system involves sending in messages such as "sign this data", to which the system responds by performing the operation if it is allowed (that is, if the key usage allows it and the key hasn't been expired via its associated certificate or revoked via a CRL and it passes whatever other checks are necessary) or returning an appropriate error code if it is disallowed. Some of the decisions made by the system can be somewhat surprising in the sense that, while valid, they come as a surprise to the user who was expecting a particular operation (for example decryption with a key for which some combination of attributes disallowed this operation) to function while the system disallowed it. This again indicates the power of the system as a whole, since it has the ability to detect problems and inconsistencies which the humans who use it would otherwise have missed.

A variation of this approach was used in the Los Alamos Advisor, an expert system which could be queried by the user to support "what-if" security scenarios with justification for the decisions reached [91]. The Advisor was first primed by rewriting a security policy originally expressed in rather informal terms such as "Procedures for identifying and authenticating users must be addressed" in the form of more precise rules

such as "IF a computer processes classified information THEN it must have identification and authentication procedures", after which it could provide advice based on the rules it had been given. The cryptlib kernel provides a similar level of functionality, although the justification for each decision which is reached currently has to be determined by stepping through the code rather than having the kernel print out the "reasoning" steps it applies.

6.3. Other Object Controls

In addition to the standard object usage access controls, the kernel can also be used to enforce a number of other controls on objects which can be used to safeguard the way in which they are used. The most critical of these is a restriction on the manner in which signing keys are used. In an unrestricted environment, a private key object, once instantiated, could be used to sign arbitrary numbers of transactions by a trojan horse or by an unauthorised outsider who has gained access to the system while the legitimate user was away or temporarily distracted. This problem is recognised by some digital signature laws, which require a distinct authorisation action (typically by entering a PIN) each time a private key is used to generate a signature. Once the single signature has been generated, the key can't be used again unless the authorisation action is performed for it.

In order to control the use of an object, the kernel can associate a usage count with it which is decremented each time the object is successfully used for an operation such as generating a signature. Once the usage count drops to zero, any further attempts to use the object are blocked by the kernel. As with the other access controls, enforcement of this mechanism is handled by decrementing the count each time an object usage message (for example one which results in the creation of a signature) is successfully processed by the object, and blocking any further messages which are sent to it once the usage count reaches zero.

Another type of control mechanism which can be used to safeguard the manner in which objects are used is a trusted authentication path, which is specific to hardware-based cryptlib implementations and is discussed in a later chapter.

7. Protecting Objects Outside the Architecture

An earlier section commented on the fact that the cryptlib security architecture contains a single trusted process-equivalent which is capable of bypassing the kernel's security controls. In cryptlib's case the "trusted process" is actually a function of half a dozen lines of code (making verification fairly trivial) which allows a key to be exported from a context in encrypted form. Normally the kernel will ensure that, once a key is present in a context, it can never be retrieved, however strict enforcement of this policy would make both key transport mechanisms which exchange an encrypted session key with another party and long-term key storage impossible. Because of this, cryptlib contains the equivalent of a trusted downgrader which allows keys to be exported from a context under carefully controlled conditions.

Although the key export and import mechanism has been presented as a trusted downgrader (because this is the terminology which is usually applied for this type of function), in reality it acts not as a downgrader but as a transformer of the sensitivity level of the key, cryptographically enforcing both the Bell-LaPadula secrecy and Biba integrity model for the keys [92].

The key export process as viewed in terms of the Bell-LaPadula model is shown in Figure 17. The key, with a high sensitivity level, is encrypted with a key encryption key (KEK), reducing it to a low sensitivity level since it's now protected by the KEK. At this point it can be moved outside the security architecture. If it needs to be used again, the encrypted form is decrypted inside the architecture, transforming it back to the high-sensitivity-level form. Since the key can only leave the architecture in a low-sensitivity form, this process isn't a true downgrading process but actually a transformation which alters the form of the high-sensitivity data to ensure that it can survive in a low-sensitivity environment.

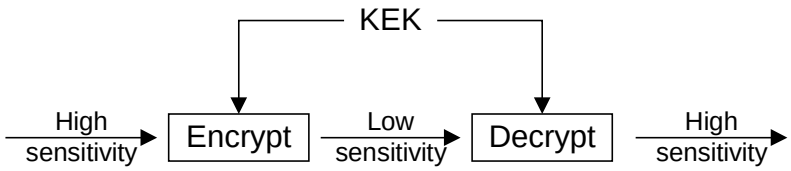

Figure 17: Key sensitivity level transformation

Although the process has been depicted as encryption of a key using a symmetric KEK, the same holds for the communication of session keys using asymmetric key transport keys.

The same process can be used to enforce the Biba integrity model using MACing, encryption, or signing to transform the data from its internal high-integrity form in a manner which is suitable for existence in the external, low-integrity environment. This process is shown in Figure 18.

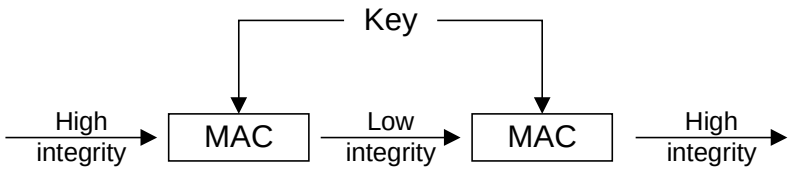

Figure 18: Key integrity level transformation

Again, although the process has been depicted in terms of MACing, it also applies for digitally signed and encrypted[5] data.

We can now look at an example of how this type of protection is applied to data when it has to leave the architecture's security perimeter. The example we'll use is a public key, which requires integrity protection but no confidentiality protection. To enforce the transformation required by the Biba model, we sign the public key (along with a collection of user-supplied data) to form a public-key certificate which can then be safely exported outside the architecture and exist in a low-integrity environment as shown in Figure 19.

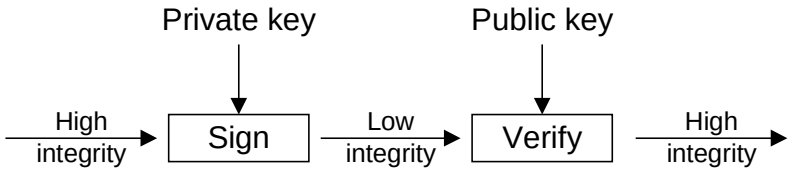

Figure 19: Public key integrity level transformation via certificate

When the key is moved back into the architecture, its signature is verified, transforming it back into the high-integrity form for internal use.

8. Object Attribute security

The discussion of security features has so far concentrated on object security features, however the same security mechanisms are also applied to object attributes. An object attribute is a property belonging to an object or a class of objects, for example encryption, signature, and MAC contexts have a key attribute associated with them, certificate objects have various validity period attributes associated with them, and device objects typically have some form of PIN attribute associated with them.

Just like objects, each attribute has an ACL which specifies how it can be used and applied, with ACL enforcement being handled by the security kernel. For example the ACL for a key attribute for a triple DES encryption context would have the entries shown in Figure 20. In this case the ACL requires that the attribute

[5] Technically speaking encryption with a KEK doesn't provide the same level of integrity protection as a MAC, however what's being encrypted with a KEK is either a symmetric session key or a private key for which an attack is easily detected when a standard key wrapping format is used.

value be exactly 192 bits long (the size of a three-key triple DES key), and it will only allow it to be written once (in other words once a key is loaded it can't be overwritten, and it can never be read). The kernel checks all data flowing in and out against the appropriate ACL, so that not only data flowing from the user into the architecture (for example identification and authentication information) but also the limited amount of data which is allowed to flow from the architecture to the user (for example status information) is carefully monitored by the kernel.

```
attribute label = CRYPT_CTXINFO_KEY
type = octet string
permissions = write-once
size = 192 bits minimum, 192 bits maximum
```

Figure 20: Triple DES key attribute ACL

Ensuring that external software can't bypass the kernel's ACL checking requires very careful design of the I/O mechanisms to ensure that no access to architecture-internal data is ever possible. Consider the fairly typical situation in which an encrypted private key is read from disk by an application, decrypted using a user-supplied password, and used to sign or decrypt data. Using techniques such as patching the systemwide vectors for file I/O routines (which are world-writeable under Windows NT) or debugging facilities like `truss` and `ptrace` under Unix, hostile code can determine the location of the buffer into which the encrypted key is copied and monitor the buffer contents until they change due to the key being decrypted, at which point it has the raw private key available to it. An even more serious situation occurs when a function interacts with untrusted external code by supplying a pointer to information located in an internal data structure, in which case an attacker can take the returned pointer and add or subtract whatever offset is necessary to read or write other information which is stored nearby. With a number of current security toolkits, something as simple as flipping a single bit is enough to turn off some of the encryption (and in at least one case turn on much stronger encryption than the US-exportable version of the toolkit is supposed to be capable of), cause keys to be leaked, and have a number of other interesting effects.

In order to avoid these problems, the architecture never provides direct access to any internal information. All object attribute data is copied in and out of memory locations supplied by the external software into separate (and unknown to the external software) internal memory locations. In cases where supplying pointers to memory is unavoidable (for example where it's required for `fread` or `fwrite`), the supplied buffers are scratch buffers which are decoupled from the architecture-internal storage space in which the data will eventually be processed.

This complete decoupling of data passing in or out means that it is very easy to run an implementation of the architecture in its own address space or even in physically separate hardware without the user ever being aware that this is the case, for example under Unix the implementation would run as a daemon owned by a different user and under Windows NT it would run as a system service. Alternatively, the implementation can run on dedicated hardware which is physically isolated from the host system as described in a later chapter.

9. References

[1] "The Protection of Information in Computer Systems", Jerome Saltzer and Michael Schroeder, *Proceedings of the IEEE*, **Vol.63**, **No.9** (September 1975), p.1278.

[2] "Object-Oriented Software Construction, Second Edition", Bertrand Meyer, Prentice Hall, 1997.

[3] "Assertion Definition Language (ADL) 2.0", X/Open Group, November 1998.

[4] "Security in Computing", Charles Pfleeger, Prentice-Hall, 1989.

[5] "Why does Trusted Computing Cost so Much", Susan Heath, Phillip Swanson, and Daniel Gambel, *Proceedings of the 14th National Computer Security Conference*, October 1991, p.644. Republished in the *Proceedings of the 4th Annual Canadian Computer Security Symposium*, May 1992, p.71.

[6] "Protection", Butler Lampson, *Proceedings of the 5th Princeton Symposium on Information Sciences and Systems*, Princeton, 1971, p.437.

[7] "Issues in Discretionary Access Control", Deborah Downs, Jerzy Rub, Kenneth Kung, and Carole Joran, *Proceedings of the 1985 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1985, p.208.

[8] "A lattice model of secure information flow", Dorothy Denning, *Communications of the ACM*, **Vol.19**. **No.5** (May 1976), p.236.

[9] "Improving Security and Performance for Capability Systems", Paul Karger, PhD Thesis, University of Cambridge, October 1988.

[10] "A Secure Identity-Based Capability System", Li Gong, *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1989, p.56.

[11] "An Analysis of Access Control Models", Gregory Saunders, Michael Hitchens, and Vijay Varadharajan, *Proceedings of the Fourth Australasian Conference on Information Security and Privacy (ACISP'99)*, Springer-Verlag Lecture Notes in Computer Science No.1587, April 1999, p.281.

[12] "Designing the GEMSOS Security Kernel for Security and Performance", Roger Schell, Tien Tao, and Mark Heckman, *Proceedings of the 8th National Computer Security Conference*, September 1985, p.108.

[13] "Secure Computer Systems: Mathematical Foundations and Model", D.Elliott Bell and Leonard LaPadula, M74-244, MITRE Corporation, 1973.

[14] "Secure Computing: The Secure Ada Target Approach", W.Boebert, R.Kain, and W.Young, *Scientific Honeyweller*, Vol.6, No.2 (July 1985).

[15] "A Note on the Confinement Problem", Butler Lampson, *Communications of the ACM*, **Vol.16**, **No.10** (October 1973), p.613.

[16] "Trusted Computer Systems Evaluation Criteria", DOD 5200.28-STD, US Department of Defence, December 1985.

[17] "Integrity Considerations for Secure Computer Systems", Kenneth Biba, ESD-TR-76-372, USAF Electronic Systems Division, April 1977.

[18] "Operating System Integrity", Greg O'Shea, *Computers and Security*, **Vol.10**, **No.5** (August 1991), p.443.

[19] "Risk Analysis of 'Trusted Computer Systems'", Klaus Brunnstein and Simone Fischer-Hübner, *Computer Security and Information Integrity*, Elseview Science Publishers, 1991, p.71.

[20] "A Comparison of Commercial and Military Computer Security Policies", David Clark and David Wilson, *Proceedings of the 1987 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1987, p.184.

[21] "Transaction Processing: Concepts and Techniques" Jim Gray and Andreas Reuter, Morgan Kaufmann, 1993.

[22] "Atomic Transactions", Nancy Lynch, Michael Merritt, William Weihl, and Alan Fekete, Morgan Kaufmann, 1994.

[23] "Principles of Transaction Processing", Philip Bernstein and Eric Newcomer, Morgan Kaufman Series in Data Management Systems, January 1997.

[24] "Non-discretionary controls for commercial applications", Steven Lipner, *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1982, p.2.

[25] "Putting Policy Commonalities to Work", D.Elliott Bell, *Proceedings of the 14th National Computer Security Conference*, October 1991, p.456.

[26] "The Chinese Wall Security Policy", David Brewer and Michael Nash, *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1989, p.206.

[27] "Lattice-Based Enforcement of Chinese Walls", Ravi Sandhu, *Computers and Security*, **Vol.11**, **No.8** (December 1992), p.753.

[28] "A Retrospective on the Criteria Movement", Willis Ware, *Proceedings of the 18th National Information Systems Security Conference* (formerly the National Computer Security Conference), October 1995, p.582.

[29] "Certification of programs for secure information flow", Dorothy Denning, *Communications of the ACM*, **Vol.20**, **No.6** (June 1977), p.504.

[30] "Computer Security: A User's Perspective", Lenora Haldenby, *Proceedings of the 2nd Annual Canadian Computer Security Conference*, March 1990, p.63.

[31] "Some Extensions to the Lattice Model for Computer Security", Jie Wu, Eduardo Fernandez, and Ruigang Zhang, Computers and Security, Vol.11, No.4 (July 1992), p.357.

[32] "Security Kernels: A Solution or a Problem", Stanley Ames Jr., *Proceedings of the 1981 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1981, p.141.

[33] "A Security Model for Military Message Systems", Carl Landwehr, Constance Heitmeyer, and John McLean, *ACM Transactions on Computer Systems*, **Vol.2**, **No.3** (August 1984), p.198.

[34] "Formal Models for Computer Security", Carl Landwehr, *ACM Computing Surveys*, **Vol. 13**, **No. 3** (September 1981), p.247

[35] "A Taxonomy of Integrity Models, Implementations, and Mechanisms", J.Eric Roskos, Stephen Welke, John Boone, and Terry Mayfield, *Proceedings of the 13th National Computer Security Conference*, October 1990, p.541.

[36] "An Analysis of Application Specific Security Policies" Daniel Sterne, Martha Branstad, Brian Hubbard, Barbara Mayer, and Dawn Wolcott, *Proceedings of the 14th National Computer Security Conference*, October 1991, p.25.

[37] "The Multipolicy Paradigm for Trusted Systems", Hilary Hosmer, *Proceedings of the 1992 New Security Paradigms Workshop*, ACM, 1992, p.19.

[38] "Metapolicies II", Hilary Hosmer, *Proceedings of the 15th National Computer Security Conference*, October 1992, p.369.

[39] "Security Kernel Design and Implementation: An Introduction", Stanley Ames Jr, Morrie Gasser, and Roger Schell, *IEEE Computer*, **Vol.16**, **No.7** (July 1983), p.14.

[40] "Kernels for Safety?", John Rushby, *Safe and Secure Computing Systems*, Blackwell Scientific Publications, 1989, p.210.

[41] "Security policies and security models", Joseph Goguen and José Meseguer, *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1982, p.11.

[42] "The Architecture of Complexity", Herbert Simon, *Proceedings of the American Philosophical Society*, **Vol.106**, **No.6** (December 1962), p.467.

[43] "Design and Verification of Secure Systems", John Rushby, *ACM Operating Systems Review*, **Vol.15**, **No.5** (December 1981), p.12.

[44] "Developing Secure Systems in a Modular Way", Qi Shi, J.McDermid, and J.Moffett, *Proceedings of the 8th Annual Conference on Computer Assurance (COMPASS'93)*, IEEE Computer Society Press, 1993, p.111.

[45] "A Separation Model for Virtual Machine Monitors", Nancy Kelem and Richard Feiertag, *Proceedings of the 1991 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1991, p.78.

[46] "A Retrospective on the VAX VMM Security Kernel", Paul Karger, Mary Ellen Zurko, Douglas Bonin, Andrew Mason, and Clifford Kahn, *IEEE Transactions on Software Engineering*, **Vol.17**, **No.11** (November 1991), p1147.

[47] "Separation Machines", Jon Graff, *Proceedings of the 15th National Computer Security Conference*, October 1992, p.631.

[48] "Proof of Separability: A Verification Technique for a Class of Security Kernels", John Rushby, *Proceedings of the 5th Symposium on Programming*, Springer-Verlag Lecture Notes in Computer Science No.137, August 1982.

[49] "A Comment on the 'Basic Security Theorem' of Bell and LaPadula", John McLean, *Information Processing Letters*, **Vol.20**, **No.2** (15 February 1985), p.67.

[50] "On the validity of the Bell-LaPadula model", E.Roos Lindgren and I.Herschberg, *Computers and Security*, **Vol.13**, **No.4** (1994), p.317.

[51] "New Thinking About Information Technology Security", Marshall Abrams and Michael Joyce, *Computers and Security*, **Vol.14**, **No.1** (January 1995), p.57.

[52] "Locking Computers Securely", O.Sami Saydari, Joseph Beckman, and Jeffrey Leaman, *Proceedings of the 10th Annual Computer Security Conference*, 1987, p.129.

[53] "Constructing an Infosec System Using the LOCK Technology", W.Earl Boebert, *Proceedings of the 8th National Computer Security Conference*, October 1988, p.89.

[54] "Programming a VIPER", T.Buckley, P.Jesty, *Proceedings of the 4th Annual Conference on Computer Assurance (COMPASS'89)*, IEEE Computer Society Press, 1989, p.84.

[55] "Report on the Formal Specification and Partial Verification of the VIPER Microprocessor", Bishop Brock and Warren Hunt Jr., *Proceedings of the 6th Annual Conference on Computer Assurance (COMPASS'91)*, IEEE Computer Society Press, 1991, p.91.

[56] "The Intel 80x86 Processor Architecture: Pitfalls for Secure Systems", Olin Sibert, Phillip Porras, and Robert Lindell, *Proceedings of the 1995 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1995, p.211.

[57] "The Segment Descriptor Cache", Robert Collins, *Dr.Dobbs Journal*, August 1998.

[58] "The Caveats of Pentium System Management Mode", Robert Collins, *Dr.Dobbs Journal*, May 1997.

[59] "Security Requirements for Cryptographic Modules", FIPS PUB 140-2, National Institute of Standards and Technology, December 1999 (draft).

[60] "Cryptographic Application Programming Interfaces (APIs)", Bill Caelli, Ian Graham, and Luke O'Connor, *Computers and Security*, **Vol.12**, **No.7** (November 1993), p.640.

[61] "The Best Available Technologies for Computer Security", Carl Landwehr, *IEEE Computer*, **Vol.16**, **No 7** (July 1983), p.86.

[62] "A GYPSY-Based Kernel", Bret Hartman, *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1984, p.219.

[63] "KSOS — Development Methodology for a Secure Operating System", T.Berson and G.Barksdale, *National Computer Conference Proceedings*, **Vol.48** (1979), p.365.

[64] "A Network Pump" , Myong Kang, Ira Moskowitz, and Daniel Lee, *IEEE Transactions on Software Engineering*, **Vol.22**, **No.5** (May 1996), p.329.

[65] "Design and Assurance Strategy for the NRL Pump", Myong Kang, Andrew Moore, and Ira Moskowitz, *IEEE Computer*, **Vol.31**, **No.4** (April 1998), p.56.

[66] "Blacker: Security for the DDN: Examples of A1 Security Engineering Trades", Clark Weissman, *Proceedings of the 1992 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1992, p.286.

[67] "Panel Session: Kernel Performance Issues", Marvin Shaefer (chairman), *Proceedings of the 1981 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1981, p.162.

[68] "AIM — Advanced Infosec Machine", Motorola Inc, 1999.

[69] "AIM — Advanced Infosec Machine — Multi-Level Security", Motorola Inc, 1998.

[70] "Formal Construction of the Mathematically Analyzed Separation Kernel", W.Martin, P.White, F.S.Taylor, and A.Goldberg, *Proceedings of the 15th International Conference on Automated Software Engineering*, IEEE Computer Society Press, September 2000, to appear.

[71] "An Avenue for High Confidence Applications in the 21st Century", Timothy Kremann, William Martin, and Frank Taylor, *Proceedings of the 22nd National Information Systems Security Conference* (formerly the National Computer Security Conference), October 1999, CDROM distribution.

[72] "Integrating an Object-Oriented Data Model with Multilevel Security", Sushil Jajodia and Boris Kogan, *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1990, p.76.

[73] "Security Issues of the Trusted Mach System", Martha Branstad, Homayoon Tajalli, and Frank Meyer, *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1988, p.362.

[74] "Access Mediation in a Message Passing Kernel", Martha Branstad, Homayoon Tajalli, Frank Meyer, and David Dalva, *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1989, p.66.

[75] "The Integrity-Lock Approach to Secure Database Management", Richard Graubart, *Proceedings of the 1984 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1984, p.62.

[76] "Towards Practical MLS Database Management Systems using the Integrity Lock Technology", Rae Burns, *Proceedings of the 9th National Computer Security Conference*, September 1986, p.25.

[77] "Protected Groups: An Approach to Integrity and Secrecy in an Object-oriented Database", James Slack and Elizabeth Unger, *Proceedings of the 15th National Computer Security Conference*, October 1992, p.513.

[78] "Security In An Object-Oriented Database", James Slack, *Proceedings of the 1993 New Security Paradigms Workshop*, ACM, 1993, p.155.

[79] "An Access Control Language for Object-Oriented Programming Systems", Masaaki Mizuno and Arthur Oldehoeft, *The Journal of Systems and Software*, **Vol.13**, **No.1** (September 1990), p.3.

[80] "Separation of Duties in Computerised Information Systems", Ravi Sandhu, *Database Security IV: Status and Prospects*, Elseview Science Publishers, 1991, p.179.

[81] "Transaction Control Expressions for Separation of Duties", Ravi Sandhu, *Proceedings of the 4th Aerospace Computer Security Applications Conference*, December 1988, p.282.

[82] "Enforcing Complex Security Policies for Commercial Applications", I-Lung Kao and Randy Chow, *Proceedings of the 19th Annual International Computer Software and Applications Conference (COMPSAC'95)*, IEEE Computer Society Press, 1995, p.402.

[83] "Enforcement of Complex Security Policies with BEAC", I-Lung Kao and Randy Chow, *Proceedings of the 18th National Information Systems Security Conference* (formerly the National Computer Security Conference), October 1995, p.1.

[84] "A TCB Subset for Integrity and Role-based Access Control", Daniel Sterne, *Proceedings of the 15th National Computer Security Conference*, October 1992, p.680.

[85] "Regulating Processing Sequences via Object State", David Sherman and Daniel Sterne, *Proceedings of the 16th National Computer Security Conference*, October 1993, p.75.

[86] "A Relational Database Security Policy", Rae Burns, *Computer Security and Information Integrity*, Elseview Science Publishers, 1991, p.89.

[87] "Extended Discretionary Access Controls", Stephen Vinter, *Proceedings of the 1988 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1988, p.39.

[88] "On the Need for a Third Form of Access Control", Richard Graubart, *Proceedings of the 12th National Computer Security Conference*, October 1989, p.296.

[89] "Beyond the Pale of MAC and DAC — Defining New Forms of Access Control", Catherine McCollum, Judith Messing, and LouAnna Notargiacomo, *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1990, p.190.

[90] "Testing Object-Oriented Systems", Robert Binder, Addison-Wesley, 1999.

[91] "Knowledge-Based Computer Security Advisor", W.Hunteman and M.Squire, *Proceedings of the 14th National Computer Security Conference*, October 1991, p.347.

[92] "Integrating Cryptography in the Trusted Computing Base", Michael Roe and Tom Casey, *Proceedings of the 1990 IEEE Symposium on Security and Privacy*, IEEE Computer Society Press, 1990, p.50.