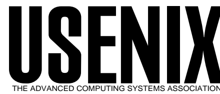
The following paper was originally published in the

Proceedings of the 8th USENIX Security Symposium

Washington, D.C., USA, August 23–26, 1999

THE DESIGN OF A CRYPTOGRAPHIC SECURITY ARCHITECTURE

Peter Gutmann



© 1999 by The USENIX Association
All Rights Reserved
For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org
Rights to individual papers remain with the author or the author's employer. Permission is granted for noncommercial reproduction of the work for educational or research purposes. This copyright notice must be included in the reproduced paper.
USENIX acknowledges all trademarks herein.

The Design of a Cryptographic Security Architecture

Peter Gutmann
University of Auckland, Auckland, New Zealand
pgut001@cs.auckland.ac.nz

Abstract

Traditional security toolkits have concentrated mostly on defining a programming interface (API) and left the internals up to individual implementors. This paper presents a design for a portable, flexible security architecture based on traditional computer security models involving a security kernel which controls access to security-relevant objects and attributes based on a configurable security policy. Layered on top of the kernel are various objects which abstract core functionality such as encryption and digital signature capabilities, certificate management, and secure sessions and data enveloping (email encryption) in a manner which allows them to be easily moved into cryptographic devices such as smart cards and crypto accelerators for extra performance or security. The versatility of the design has been proven through its use in implementations ranging from from 16-bit microcontrollers through to supercomputers, as well as a number of unusual areas such as security modules in ATM's.

1. Introduction

Traditionally, security toolkits have been implemented using a "collection of functions" design in which each encryption capability is wrapped up in its own set of functions. For example there might be a "load a DES key" function, an "encrypt with DES in CBC mode" function, a "decrypt with DES in CFB mode" function, and so on [1][2]. More sophisticated toolkits hide the plethora of algorithm-specific functions under a single set of umbrella interface functions with often complex algorithm-selection criteria, in some cases requiring the setting of up to a dozen parameters to select the mode of operation [3][4][5][6]. Either approach requires that developers tightly couple the application to the underlying encryption implementation, requiring a high degree of cryptographic awareness from developers and forcing each new algorithm and application to be treated as a distinct development.

Alternative approaches concentrate on providing functionality for a particular type of service such as authentication, integrity, or confidentiality. An example of this type of design is the GSS-API [7], which is session-oriented and is used to control session-style communications with other entities (an example implementation consists of a set of GSS-API wrapper functions for Kerberos), the OSF DCE security API [8], which is based around ACL's and secure RPC, and the SESAME API [9] which is based around a Kerberos derivative with various enhancements such as X.509 certificate support. This type of design typically includes features specific to the required functionality, in the case of the session-oriented interfaces mentioned above this is the security context which contains details of a relationship between peers based on credentials established between the peers. A non-session-based variant is the IDUP-GSS-API [10], which attempts to stretch the GSS-API to cover store-and-forward use (this would typically be used for a service such as email protection).

Both of these approaches represent an outside-in approach which begins with a particular programming interface and then bolts on whatever is required to implement the functionality in the interface. This paper presents an alternative inside-out design which takes a general crypto/security architecture and then wraps a language-independent interface around it to make particular portions of the architecture available to the user. In this case it is important to distinguish between the architecture and the API used to interface to it — with most approaches the API *is* the architecture, whereas the approach presented in this paper concentrates on the internal architecture only. Apart from the very generic APKI requirements [11], only CDSA [12] appears to provide a general architecture design, and even this is presented at a rather abstract level and defined mostly in terms of the API used to access it.

In contrast to these approaches, the design presented in this paper begins by defining the general requirements for an object model upon which to build the architecture, which is used to encapsulate various types of functionality such as encryption and certificate management. The first portion of the paper presents the overall design goals for the architecture, as well as the details of each object class. Since the entire architecture has very stringent security requirements, the object model requires an underlying security kernel capable of supporting it, one which includes a means of mediating access to objects, controlling the way this access is performed (for example the manner in which

object attributes may be manipulated), and ensuring strict isolation of objects (that is, ensuring that one object can't influence the operation of another object in an uncontrolled manner). The security aspects of the architecture are covered in the second part of the paper.

2. Architecture Design Goals

An earlier work [13] gives the design requirements for a general-purpose API, including algorithm, application, and cryptomodule independence, safe programming (protection against programmer mistakes), a security perimeter to prevent sensitive data from leaking out into untrusted applications, and legacy support. Most of these requirements are pure API issues and won't be covered in any more detail here. The architecture presented here is built on the following design principles:

- Independent objects. Each object is responsible for managing its own resource requirements such as memory allocation and use of other required objects, and the interface to other objects is handled in an object-independent manner. For example a signature object would know that it is associated with a hash object, but wouldn't need to know any details of it's implementation such as function names or parameters in order to communicate with it. In addition each object has associated with it various security properties such as mandatory and discretionary ACL's, some of which are controlled for the object by the architectures security kernel and some object-specific properties which are controlled by the object itself.
- Intelligent objects. The architecture should know what to do with data and control information passed to objects, including the ability to hand it off to other objects where required. For example if a certificate object (which contains only certificate-related attributes but has no inherent encryption or signature capabilities) is asked to verify a signature using the key contained in the certificate, the architecture will hand the task off to the appropriate signature checking object without the user having to be aware that this is occurring. This leads to a very natural interface in which the user knows an object will Do The Right Thing with any data or control information sent to it, without requiring it to be accessed or used in a particular manner.
- Platform-independent design. The entire architecture should be easily portable to a wide variety of hardware types and operating systems without any significant loss of functionality. A counterexample to this design requirement is CryptoAPI 2.x [14], which is so heavily tied into features of the very newest versions of Win32 that it would be almost impossible to move to other platforms. In contrast the architecture described here was designed from the outset to be extremely portable and has been implemented on everything from 16-bit microcontrollers with no filesystem or I/O capabilities through to supercomputers, as well as unconventional designs like multiprocessor Tandem machines and IBM VM/ESA mainframes.
- Full isolation of architecture internals from external code. The architecture internals are fully decoupled from access by external code, so that the implementation may reside in its own address space (or even physically separate hardware) without the user being aware of this. The reason for this requirement is that it very clearly defines the boundaries of the architecture's trusted computing base (TCB), allowing the architecture to be defined and analysed in terms of traditional computer security models.
- Layered design. The architecture represents a true object-based multilayer design, with each layer of functionality built on its predecessor. The purpose of each layer is to provide certain services to the layer above it, shielding that layer from the details of how the service is actually implemented. Between each layer is an interface which allows data and control information to pass across in a controlled manner. In this way each layer provides a set of well-defined and understood functions which both minimise the amount of information which flows from one layer to another, and makes it easy to replace the implementation of one layer with a completely different implementation (for example migrating a software implementation into secure hardware), because all that a new layer implementation requires is that it offer the same service interface as the one it replaces.

In addition to the layer-based separation, the architecture separates individual objects within the layer into discrete, self-contained objects which are independent of other objects both within their layer and in other layers. For example in the lowest layer the basic objects typically represent an instantiation of a single encryption, digital signature, key exchange, hash, or MAC algorithm. Each object can represent a software implementation, a hardware implementation, a hybrid of the two, or some other implementation.

3. The Object Model

The architecture implements two types of objects, container objects and action objects. A container object is an object which contains one or more items such as data, keys, certificates, security state information, and security attributes. The container types can be broken down roughly into three types, data containers (referred to as envelope or session objects), key and certificate containers (keyset objects), and security attribute containers (certificate objects). An action object is an object which is used to perform an action such as encrypting, hashing, or signing data (referred to as an encryption context). Action objects are fairly simple and encapsulate the functionality of a security algorithm such as DES or RSA, these function mainly as building blocks used by the more complex object types. In addition to these standard object types, there is also a device object type which constitutes a meta-object used to work with external encryption devices such as smart cards or Fortezza cards which may require extra functions such as activation with a user PIN before they can be used. Once they're initialised as required, they can be used like any of the other object types whose functionality they provide, for example an RSA action object could be created through the device object for a smart card with RSA capabilities, or a certificate object could be stored in a device object for a Fortezza card as if it were a keyset.

The implementation of each object is completely hidden from the user, so that the only way they can access the object is by sending information to it across a carefully-controlled channel. Figure 1 illustrates how three low-level action objects (implementing DES, SHA-1, and RSA) would be handled. The object handles are small integer values, unrelated to the object itself, which are used to pass control information and data to and from the object. Since each object is referred to through an abstract handle, the interface to the object is a message-based one in which messages are sent to and received from the object. Although the external programming interface can be implemented to look like the traditional "collection of functions" one, this is simply the message-passing interface wrapped up to look like a more traditional functional interface.

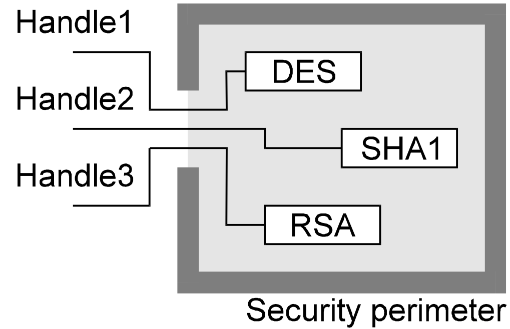


Figure 1. Typical low-level objects

Container objects generally contain other objects (as well as data and attributes) within them. For example each certificate object has an (internal) public or private key context attached to it as shown in Figure 2.

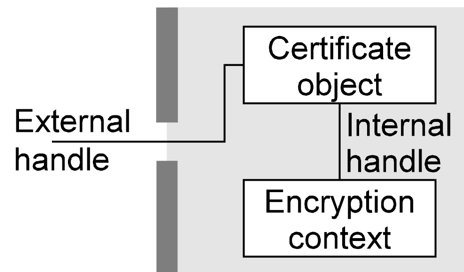


Figure 2. Object with dependent object

This encryption context can't be directly accessed by the user, but can be used in the carefully controlled manner provided by the certificate object. For example if the certificate object contains an attribute specifying that the attached public-key context may only be used for digital signature (but not encryption) purposes then any attempt to use the object for encryption purposes would be flagged as an error.

The three types of container object differ mainly in the view they present to the user, and are explained below.

3.1. Data Containers

Data containers (envelope and session objects) are objects whose behaviour is modified by the data and attributes which are pushed into them. To use an envelope, the user pushes in control information in the form of container or action objects or general attributes which control the behaviour of the container. Any data which is pushed into the container is then modified according to the behaviour established by the control information. For example if a digital signature action object was added to the data container as control information then data pushed into the container would be digitally signed; if a password attribute was pushed

into the container then data pushed in would be encrypted.

Session objects function in a similar manner, but the security context for the session is usually established by exchanging information with a peer, and the session objects can process multiple data objects (for example network packets) rather than the single data object processed by envelopes — session objects are envelope objects with state. In real-world terms, envelope objects would be used for functions like S/MIME and PGP while session objects would be used for functions like SSL and ssh.

This type of object can be regarded as an intelligent container which knows how to handle data provided to it based on control information it receives. For example if the user pushes in a password attribute followed by data, the object knows that the presence of this attribute implies a requirement to encrypt data and will therefore create an encryption action object, turn the password into the appropriate key type for the object (typically through the use of a hash action object), generate an initalisation vector, pad the data out to the cipher block size if necessary, encrypt the data, and return the encrypted result to the user. Data containers, although appearing relatively simple, are by far the most complex objects present in the architecture.

3.2. Key and Certificate Containers

Key and certificate containers (keyset objects) are simple objects which contain one or more public or private keys or certificates, and may contain additional information such as key revocation data (CRL's). To the user, they appear as an (often large) collection of encryption contexts or certificate objects. Two typical container objects of this type are shown in Figure 3. Although the diagram implies the presence of huge numbers of objects, these are only instantiated when required by the user. Keyset objects are tied to whatever underlying storage mechanism is used to hold keys, typically flat files, PGP keyrings, relational databases containing certificates and CRL's, LDAP directories, or HTTP links to certificates published on web pages.

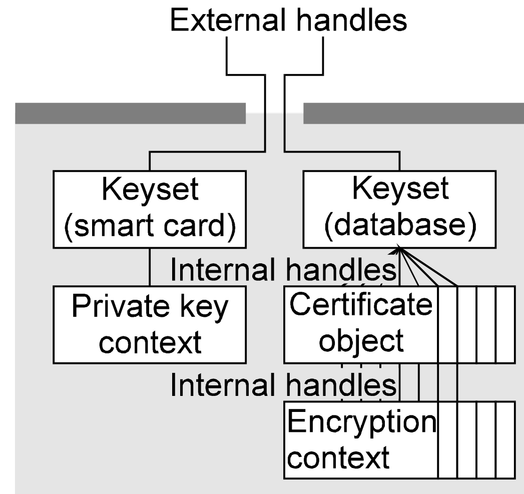


Figure 3. Container objects

3.3. Security Attribute Containers

Security attribute containers (certificate objects) contain a collection of attributes attached to a public/private key, or attributes which are attached to other information (for example signed data often comes with accompanying attributes such as the signing time and details on the signer of the data and the conditions under which the signature was generated). The most common type of security attribute container is the key certificate, which contains attribute information for a public or private key.

4. Security Features of the Architecture

Because of the lack of inter- and intra-process security present in a number of widely-deployed systems, it becomes necessary for the architecture to provide its own object security mechanisms. 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). Although full coverage of the various security models and requirements is beyond the scope of this paper, one typical source [15] provides a set of requirements for a security architecture which are implemented in the cryptlib security toolkit [16] in the following manner:

- 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 [17], making the kernel very amenable to testing using mechanical verifiers such as ADL [18].
- 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 [19]:

- 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 security kernel which performs these functions 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. An example of the final architectural model is shown in Figure 4, which illustrates the connection between the user application and architecture objects, with the light grey lines representing items with implicit connections (the kernel to the ACL's and one object to another, typically a certificate object to an associated encryption context.

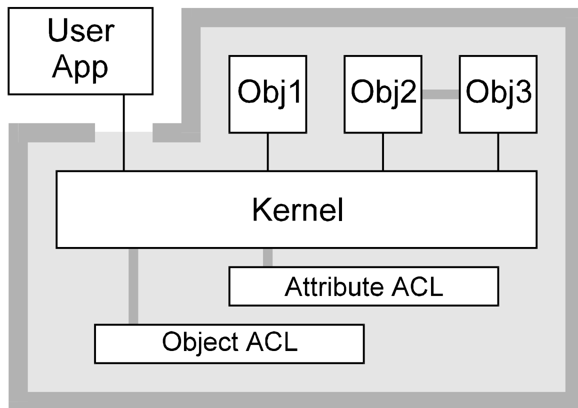


Figure 4. Architecture security model

5. Object Security and Access Control

The most important security feature of the architecture is that each object 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. For example once keying information has been sent to an object, it can't be retrieved by the user except under tightly-controlled conditions (the only real case where this can occur is when an object's ACL permits a short-term session key to be exported in encrypted form, or a private key to be stored in encrypted form to a permanent storage medium such as a smart card or disk). In general keying information isn't even visible to the user, since it is generated inside the object itself and never leaves the security perimeter. This design is ideally matched to hardware implementations which perform strict red/black separation, since sensitive information can never leave the hardware.

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. At a very coarse level, each object has a mandatory access control setting which determines whether it is externally visible or not (that is, whether it has a handle which is valid outside the security perimeter). Only externally visible objects can be (directly) manipulated by the user, with ACL enforcement being handled by the architectures security kernel.

Another ACL entry 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 may 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 5, 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.

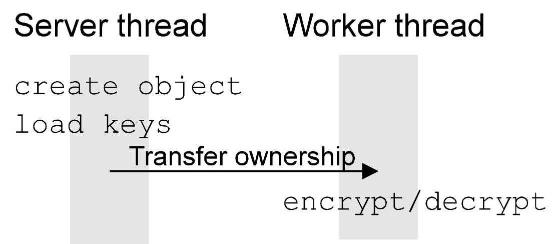


Figure 5. Object ownership transfer

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[1]. 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. The usual way of implementing this fine level of control is to use a security attribute object containing the control information attached to an encryption context. Enforcement of finer-grained attribute-based DACL settings is handled by the object itself, since these settings are specific to each object type.

ACL's are inherited across objects, so that retrieving a private key encryption object from a keyset container

[1] 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.

object will copy the container object's ACL across to the private key encryption object.

5.1. Object Security Implementation

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 objects 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. Both the object table and the object data are protected through locking and ACL mechanisms. Creating a new object works as follows:

```
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
```

This simply 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 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 the architecture itself, the user never has access to this level of functionality). 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 objects state to normal, unlocks the object and returns its handle to the user.

The object table is maintained by the security kernel. When a new object is created, it tries to allocate a handle into the object table, with the handles being 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 users 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. When an object is created, the kernel sets an ACL entry which marks 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). The object can also have a variety of attributes specified for its creation such as the type of memory used (some systems can allocate limited amounts of protected, nonpageable memory which is preferred for sensitive data such as encryption contexts).

When the user passes an objects handle to cryptlib, it performs the following actions:

```
lock object table;
verify that the handle is valid;
verify that the object allows this type of
   operation;
verify that the ACL allows external access;
verify that the ACL allows access by the
   calling thread;
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
```

This performs the necessary ACL checking for the object in an object-independent manner. The link from external handles through the cryptlib-wide object table and ACL check to the object itself is shown in Figure 6.

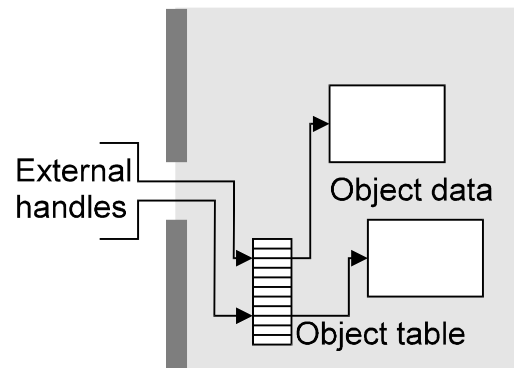


Figure 6. Object ACL checking

DACL checking is object-specific and is performed by the object itself once the basic ACL check is passed as shown in Figure 7. Architecture-internal objects are checked in a similar manner, except that the check of the external access ACL is omitted.

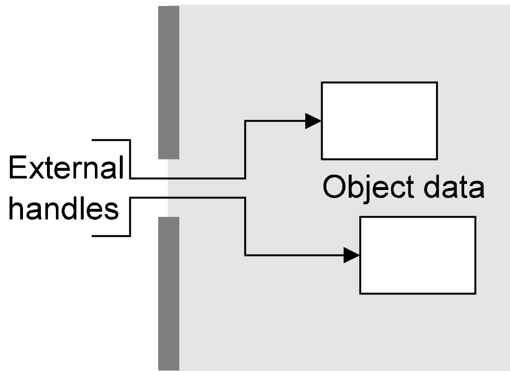


Figure 7. Object DACL checking

The ACL check is performed each time an object is used, and the ACL 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. In contrast cryptlib 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 8, 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.

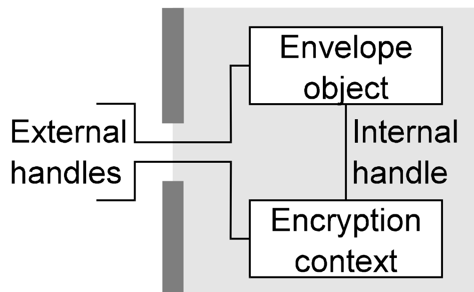


Figure 8. Objects with multiple references

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 used to make the architecture thread-safe, and tracks the identity of the thread which owns the resource. By setting the thread access ACL, a thread can claim an unowned 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.

Making each object thread-safe and providing an ACL capability across multiple operating systems is somewhat tricky. The locking and ACL capabilities in cryptlib are implemented as a collection of preprocessor macros which are designed to allow them to be mapped to appropriate OS-specific user- and system-level thread synchronisation and locking functions. Great care has been taken to ensure that this locking mechanism is as fine-grained as possible, with locks typically covering no more than a dozen or so lines of code before they are relinquished, and the code executed while the lock is active being carefully scrutinised to ensure it can never become the cause of a bottleneck, for example by executing a long-running loop while the lock is active.

5.2. 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 following entries:

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

In this case the ACL requires that the attribute 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.

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.

5.3. Benefits of the object security model

This particular object security model has several advantages. By combining the locking which is required to make the objects thread-safe with part of the ACL functionality, very little extra overhead is introduced (of the two most common operating systems with threading capabilities, both Unix user-level pthreads implementations and Windows pseudocritical sections don't usually require any kernel calls, making them relatively quick). In addition since each object is inherently thread-safe and ACL-protected, different parts of cryptlib can use and communicate with the objects without having to worry about access checking and problems with simultaneous access to shared resources — the kernel and objects take care of this themselves. Since attribute checking is also performed using ACL's rather than the traditional ad hoc parameter checks hardcoded into miscellaneous functions in various places, all attributes can have a coordinated security policy applied to them through a central, easily-checked set of ACL's.

6. Object Internals

Creating or instantiating a new object involves obtaining a new handle, allocating and initialising an internal data structure which stores information on the object, setting ACL's, connecting the object to any underlying hardware or software if necessary (for example establishing a session with a smart card reader or database backend), and finally returning the object's handle to the user. Although the user sees a single object type which is consistent across all computer systems and implementations, the exact (internal) representation of the object can vary considerably. In the simplest case, an object consists of a thin mapping layer which translates calls from the architectures's internal API to the API used by a hardware implementation. Since encryption contexts, which represent the lowest level in the architecture, have been designed to map directly onto the functionality provided by common hardware crypto accelerators, these can be used directly when appropriate hardware is present in the system.

If the encryption hardware consists of a crypto device with a higher level of functionality or even a general-purpose secure coprocessor rather than just a simple crypto accelerator, more of the functionality can be offloaded onto the device or secure coprocessor. For example while a straight crypto accelerator may support

functionality equivalent to basic DES and RSA operations on data blocks, a crypto device such as a PKCS #11 token would provide extended functionality including the necessary data formatting and padding operations required to perform secure and portable key exchange and signature operations, and more sophisticated secure coprocessors which are effectively scaled-down PC's [20] can take on board architecture functionality at an even higher level. Figure 9 shows the levels at which external hardware functionality can be integrated, with the lowest level corresponding to the functionality embodied in an encryption context, while the higher levels correspond to functionality in envelope and certificate objects.

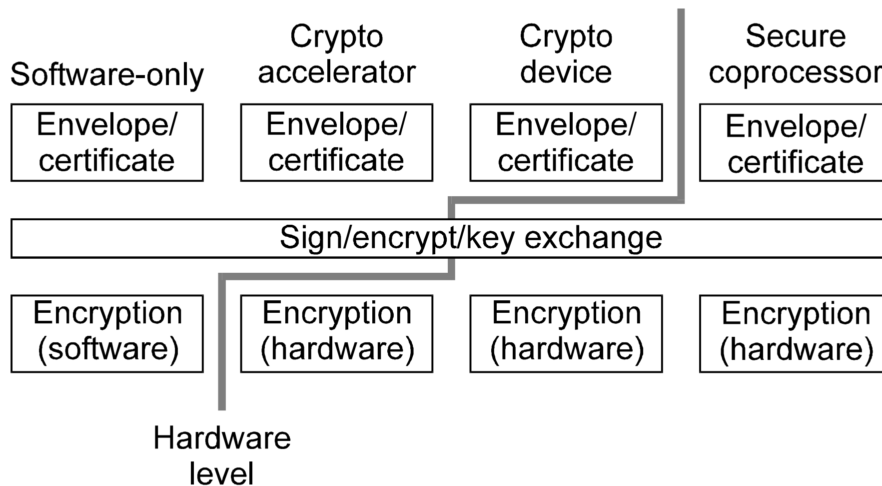


Figure 9. Mapping of architecture functionality levels to crypto/security hardware

6.1. Object Internal Details

Although each type of object differs considerably in its internal design, they all share a number of common features which will be covered here. Each object consists of three main parts:

1. State information, stored either in secure or general-purpose memory depending on its sensitivity.
2. The object's message handler.
3. A set of function pointers for the methods used by the object.

The actual functionality of the object is implemented through the function pointers, which are initialised when the object is instantiated to refer to the appropriate methods for the object. Using an instantiation of a DES encryption context with an underlying software implementation and an RSA encryption context with an underlying hardware implementation, we have the encryption context structures shown in Figure 10.

When the two objects are created, the DES context is plugged into the software DES implementation and the RSA context is plugged into a hardware RSA accelerator. Although the low-level implementations are very different, both are accessed through the same methods, typically `context.loadKey()`, `context.encrypt()`, and `context.decrypt()`. Substituting a different implementation of an encryption algorithm (or adding an entirely new algorithm) requires little more than creating the appropriate interface methods to allow a context to be plugged into the underlying implementation. As an example of how simple this can be, when the Skipjack algorithm was declassified [21] it took only a few minutes to plug in an implementation of the algorithm, providing full support for Skipjack throughout the entire architecture and to all applications which employed the architecture's standard capability query mechanism, which automatically establishes the available capabilities of the architecture on startup.

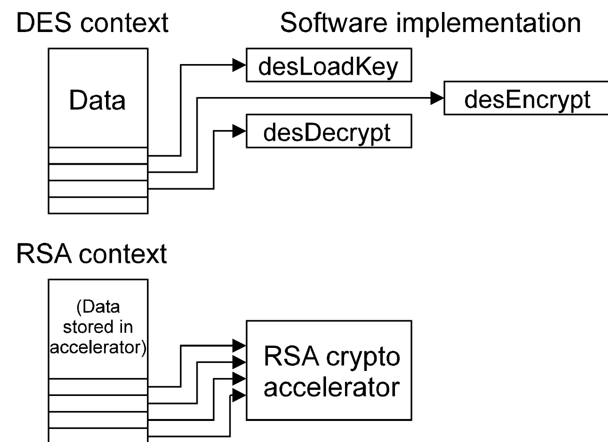


Figure 10. Encryption context internal structure

6.2. Object Reuse

Since object handles are detached from the objects they are associated with, a single object can (provided its ACL's allow this) be used by multiple processes or threads at once. This flexibility is particularly important with objects used in connection with container objects, since replicating every object pushed

into a container creates both unnecessary overhead and increases the chances of compromise of sensitive information if keys and other data are copied across to each newly created object.

Instead of copying each object whenever it is reused, the architecture maintains a reference count for it and only copies it when necessary. In general the copying is only needed for some encryption contexts, which employ a copy-on-write mechanism which ensure the object isn't replicated unnecessarily. Other objects which can't (easily) be replicated, or which don't need to be replicated, have their reference count incremented when they are reused, and decremented when they are freed. The object itself is only destroyed when its reference count drops to zero.

To see how this works, let's assume the user creates an encryption context and pushes it into an envelope object. This results in a context with a reference count of 2, with one external reference (by the user) and one internal reference (by the envelope object) as shown previously in Figure 8. Typically the user would then destroy the encryption context while continuing to use the envelope which it is now associated with. The reference with the external access ACL would be destroyed and the reference count decremented by one, leaving the object as shown in Figure 11 with a reference count of 1 and an internal access ACL.

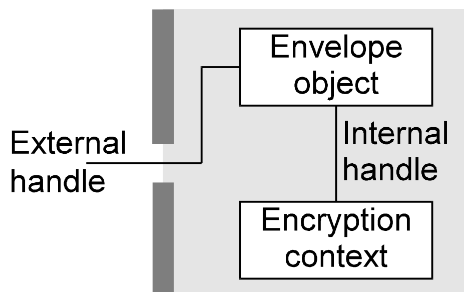


Figure 11. Objects with multiple references after the external reference is destroyed

To the user the object has indeed been destroyed, since it is now accessible only to the envelope object. When the envelope object is destroyed the encryption context's reference count is again decremented through a message sent from the envelope, leaving it at zero whereupon the kernel sends it a "destroy object" message to notify it to shut itself down, after which it removes the object from the object table. The only time objects are explicitly destroyed is through an external signal such as a smart card withdrawal or when the kernel broadcasts destroy object messages when it is closing down. At any other time only their reference count is decremented.

In some cases an object has to change its behaviour when it is reused. For example some key databases don't handle multiple independent writes at all well (this can upset their internal buffering and state management), so the object will change its ACL from read/write to read-only when its reference count becomes greater than one. Once it drops back to one, the ACL is updated to allow read/write access again.

The use of the reference-counting implementation allows objects to be treated in a far more flexible manner than would otherwise be the case. For example the paradigm of pushing attributes and objects into envelopes (which could otherwise be prohibitively expensive due to the overhead of making a new copy of the object for each envelope) is rendered feasible since in general only a single copy of each object exists. Similarly, a single (heavyweight) connection to a key database can be shared across multiple threads or processes, an important factor in a number of client/server databases where a single client connection can consume a megabyte or more of memory.

6.3. Data Formats

Since each object represents an abstract security concept, none of them are tied to a particular underlying data format or type. For example an envelope could output the result of its processing in the data format used by CMS/S/MIME, PGP, PEM, MSP, or any other format required. As with the other object types, when the envelope object is created, its function pointers are set to encoding or decoding methods which handle the appropriate data formats. In addition to the variable, data format-specific processing functions, envelope and certificate objects employ data recognition routines which will automatically determine the format of input data (for example whether data is in CMS/S/MIME or PGP format, or whether a certificate is a certificate request, certificate, PKCS #7 certificate chain, CRL, or other type of data) and set up the correct processing methods as appropriate.

7. Interobject Communications

Objects communicate internally via a message-passing mechanism, although this is typically hidden from the user by a more conventional functional interface. The message-passing mechanism connects the objects indirectly, replacing pointers and direct function calls and is the fundamental mechanism used to implement the complete isolation of architecture internals from the outside world. Since the mechanism is anonymous, it reveals nothing about an objects implementation or its interface, or even its existence. The message-passing mechanism has three parts, the source object, the

destination object, and the message dispatcher. In order to send a message from a source to a destination, the source object needs to know the target objects handle, but the target object has no knowledge of where a message came from unless the source explicitly informs it of this. All data communicated between the two is held in the message itself.

To handle interobject messaging, the kernel contains a message dispatcher which maintains an internal message queue used to forward messages to the appropriate object or objects. Some messages are directed at a particular object (identified by the objects handle), others to an entire class of object or even to all objects. For example if an encryption context is instantiated from a smart card and the card is then withdrawn from the reader, the event handler for the keyset object associated with the reader broadcasts a card withdrawal message identifying the card which was removed to all active objects as illustrated in Figure 12. This is necessary to notify the encryption context that it may need to take action based on the card withdrawal, and also to notify further objects such as envelope objects and certificates which have been created or acted upon by the encryption context — since the sender is completely disconnected from the receiver, it needs to broadcast the message to all objects to ensure that everything which might have an interest is notified. The message handler has been designed so that processing a message of this type has almost zero overhead compared to the complexity of tracking which message might apply to which objects, so it makes more sense to handle the notification as a broadcast rather than maintaining per-object lists of messages the object is interested in.

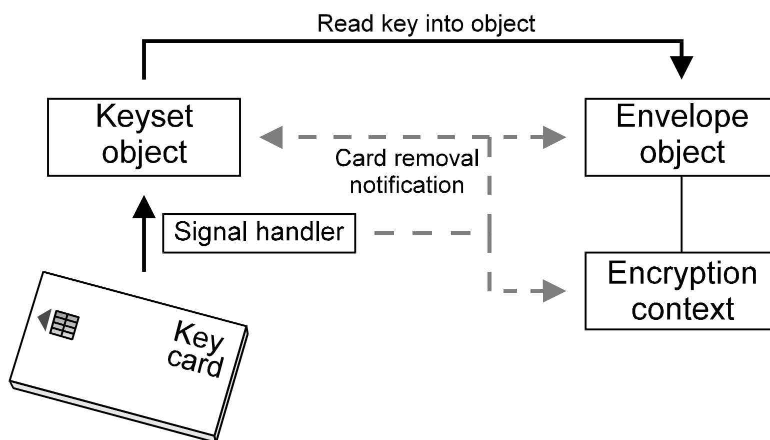


Figure 12. Interobject messaging example

As each object receives a message, it can explicitly choose to act on it or to take the default action of ignoring it. Each object therefore has the ability to intelligently handle external events in a controlled manner, processing them as appropriate. Because an object controls how it handles these events, there's no need for any other object or control routine to know about the internal details or function of the object — it simply posts a notification of an event and goes about its business.

As an example of how intelligent message fowarding across objects can work, consider an attempt to encrypt data using a certificate (in fact it's really encrypted using the encryption context associated with the certificate, but this is invisible to the user, all they see is the certificate object). This fails in some way and returns a "public-key encryption operation failed" error to the caller, who tries to get more information about what went wrong by reading the objects error information attributes. The kernel sends the certificate object a query message, but it isn't involved with the encryption operation and returns the query to the kernel with a "not at this address, try attached objects" comment, to which the kernel forwards the message to the attached encryption context. The context in turn doesn't know the exact details and returns the message to the kernel, which determines that the context is tied to some sort of underlying encryption device, so it forwards the message to the device object. The buck finally stops at the device object (which happens to be tied to a smart card which has just failed in some way), which returns the appropriate error information from the card. This demonstrates the "intelligent object" design in which an object can instruct the kernel to hand a task off to another object if it isn't capable of fulfilling the request itself — the caller ends up with telemetry from the smart card even though the object they're explicitly using is a certificate.

In the case of the card withdrawal notification illustrated in Figure 12, the affected objects which don't choose to ignore it would typically erase any security-related information, close active OS services such as open file handles, free allocated memory, and place themselves in a signalled state in which no further use of the object is possible apart from destroying it. Message queueing and dispatching is handled by the kernel's message dispatcher and the message handlers built into each object, which remove from the user the need to check for various special-case conditions such as smart card withdrawals. In practice the only object which would process the message is the encryption context. Other objects which might contain the context (for example an envelope or certificate object) will only notice the card

withdrawal if they try to use the context, at which point it will inform them that it has been signalled and is no longer useable.

Since the objects act independently, the fact that one object has changed state doesn't affect any of the other objects. This object independence is an important feature, since it doesn't tie the functioning of one object to every component object it contains or uses — a smart card-based private key might only be needed to decrypt a session key at the start of a communications session, after which its presence is irrelevant. Since each object manages its own state, the fact that the encryption context created from the key on the card has become signalled doesn't matter to the object using it after it has recovered the session key.

7.1. Asynchronous vs Synchronous Message Dispatching

When processing messages, the dispatcher can handle them in one of two ways:

1. Asynchronously, returning control to the caller immediately while processing the object in a separate thread
2. Synchronously, suspending the caller while the message is processed.

There are two types of messages which can be sent to an object, simple notifications and data communications which are processed immediately, and more complex, generally object-specific messages which can take awhile to process, an example being "generate a key" which can take awhile for many public-key algorithms. This would in theory require both types of message dispatching, however in the cryptlib architecture each object is responsible for its own handling of asynchronous processing. In practice this means that (on systems which support it) the object has one or more threads attached to it which perform asynchronous processing (on non-threaded systems there's no choice but to use synchronous messaging). When a source object sends a message to a destination which may take some time to generate a result, it includes in the message its own handle to which the destination sends a response when it's ready. When the destination object has the data required by the source ready, it sends a message back to the source object containing the data. Since the objects are inherently thread-safe, the messaging mechanism is also safe when asynchronous processing is taking place.

For an example of how the handling of messaging in an asynchronous manner works, consider a status query being sent to a keyset object connected to a keyset with a slow response time (for example a smart card or a remote LDAP server). First, the source object would send a status query message, including in the message its object handle (the return address for the query result). The dispatcher would forward the message to the keyset object, which would remember the return address and return control to the dispatcher, which would in turn return control to the source object. Some time later (whenever the keyset object has the requested information), it would send a message back to the source object containing the information it had requested.

7.2. The Message Dispatcher

The message dispatcher maintains a queue of all pending messages due to be sent to target objects which are dispatched in order of arrival. If an object isn't busy processing an existing message, a new message intended for it is immediately dispatched to it without being enqueued, which prevents the single message queue from becoming a bottleneck. For group messages (messages sent to all objects of a given type) or broadcast messages (messages sent to all objects), the message is sent to every applicable object in turn.

Recursive messages (ones which result in further messages being sent to the source object) are handled by having the dispatcher enqueue messages intended for an object which already has a message present in the queue and return immediately to the caller. This ensures that the new message isn't processed until the earlier message(s) for the object have been processed. If the message is for a different object, it is either processed immediately if the object isn't already processing a message or it is prepended to the queue and processed before other messages, so that messages sent by objects to subordinate objects are processed before messages for the objects themselves. An object won't have a new message dispatched to it until the current one has been processed. This processing order ensures that messages to the same object are processed in the order sent, and messages to different objects arising from the message to the original object are processed before the message for the original object completes.

Since an earlier message can result in an object being destroyed, the dispatcher also checks to see whether the object still exists in an active state. If not, it dequeues all further messages without calling the objects message handler.

8. Other Kernel Mechanisms

In order to work with the objects described so far, the architecture requires a number of other mechanisms to

handle synchronisation, background processing, and the reporting of events within the architecture to the user. These mechanisms are described below.

8.1. Semaphores

In the message-passing example given earlier, the source object may want to wait until the data it requested becomes available. In general since each object can potentially operate asynchronously, cryptlib requires some form of synchronisation mechanism which allows an object to wait for a certain event before it continues processing. The synchronisation is implemented using lightweight internal semaphores, which are used in most cases (in which no actual waiting is necessary) before falling back to the often heavyweight OS semaphores.

cryptlib provides two types of semaphores, system semaphores (that is, predefined semaphore handles corresponding to fixed resources or operations such as binding to various types of drivers which takes place on startup) and user semaphores, which are allocated by an object as required. System semaphores have architecture-wide unique handles akin to the stdio libraries predefined stdin, stdout, and stderr handles.

8.2. Threads

The independent, asynchronous nature of the objects in the architecture means that, in the worst case, there can be dozens of threads all whirring away inside cryptlib, most of which will be blocked while waiting on external events. Since this acts as a drain on system resources, can negatively affect performance (some operating systems can take some time to instantiate a new thread), and adds extra implementation detail for handling each thread, cryptlib provides an internal service thread which can be used by objects to perform basic housekeeping tasks. Each object can register service functions with this thread which are called in a round-robin fashion, after which the thread goes to sleep for a preset time interval, behaving much like a fiber or lightweight, user-scheduled thread. This means that simple tasks such as basic status checks can be performed by a single architecture-wide thread instead of requiring one thread per object. This service thread also performs general tasks such as touching each allocated memory page which is marked as containing sensitive data whenever it runs in order to reduce the chances of the page being swapped out.

Consider an example of a smart card keyset object which needs to check the card status every now and then to determine whether the card has been removed from the reader. Most serial-port based readers don't provide any useful notification mechanism, but only report a "card removed" status on the next attempt to access it. This isn't terribly useful to the architecture, which expects to be able to destroy objects which depend on the card as soon as it is removed.

In order to check for card removal, the keyset object registers a service function with the service thread. The registration returns a unique service ID which can be used later to deregister it. Deregistration can also occur automatically when the object which registered the service function is destroyed.

Once a service function is registered, it is called whenever the service thread runs. In the case of the keyset object it would query the reader to determine whether the card was still present. If the card is removed, it sends a message to the keyset object (running in a different thread), after which it returns, and the next service function is processed. In the meantime the keyset object notifies all dependent objects and destroys itself, in the process deregistering the service function. As with the message processing, since the objects involved are all thread-safe, there are no problems with synchronisation (for example the service function being called can deregister itself without any problems).

8.3. Event Notification

A common method for notifying the user of events is to use one or more callback functions. These functions are registered with a program and are called when certain events occur. Typical implementations use either event-specific callbacks (so the user can register functions only for events they're specifically interested in) or umbrella callbacks which get passed all events, with the user determining whether they want to act on them or not.

Callbacks have two main problems. The first of these is that they are inherently language and often OS-specific, often occurring across process boundaries and always requiring special handling to set up the appropriate stack frames, ensure arguments are passed in a consistent manner, and so on. Language-specific alternatives to callbacks such as Visual Basic event handlers are even more problematic.

The second problem with callbacks is that the called user code is given the full privileges of the calling code unless special steps are taken [22]. One possible workaround is to perform callbacks from a special no-privileges thread, but this means that the called code is given too few privileges rather than too many.

A better solution which avoids both the portability and security problems of callbacks is to avoid them altogether in favour of an object polling mechanism.

Since all encryption functionality is provided in terms of objects, object status checking is provided automatically by the security kernels reference monitor — if any object has an abnormal status associated with it (for example it might be busy performing a long-running operation such as a key generation), any attempt to use it wil result in the status being returned without any action being taken.

Because of the object-based approach used for all security functionality, the object status mechanism works transparently across arbitrarily linked objects. For example if the encryption object in which the key is being generated is pushed into an envelope, any attempt to use it before the key generation has completed will result in an "object busy" status being passed back up to the user. Since it's the encryption object which is busy (rather than the envelope), it's still possible to use the envelope for non-encryption functions while the key generation is occurring in the encryption object.

9. Conclusion

This paper has presented a flexible, platform-independent cryptographic security architecture which is suited to software, hardware, and hybrid implementations. By encapsulating the functionality inside independent intelligent objects protected by a central security kernel, portions of the architecture can be replaced or updated with a minimum of effort while guaranteeing a consistent interface and handling of the objects within the architecture. As implemented in cryptlib, this design has been successfully deployed on systems ranging from 16-bit microcontrollers through to supercomputers, languages ranging from C/C++ through to Perl and Visual Basic, and interfaced to a wide variety of cryptographic hardware and other devices, providing a single consistent interface across all of these platforms and languages (write once, encrypt anywhere).

10. Acknowledgements

The author would like to thank Peter Fenwick, Trent Jaeger, Paul Karger, and the referees for feedback and comments, and would also like to acknowledge Microsoft, whose operating system security motivated many of the design features of the architecture presented in this paper.

11. References

[1] libdes, `http://www.cryptsoft.com/ssleay/faq.html`, 1996.

[2] "Fortezza Cryptologic Programmers Guide", Version 1.52, National Security Agency Workstation Security Products, 30 January 1996.

[3] "BSAFE Library Reference Manual", Version 4.0, RSA Data Security, 1998.

[4] "Generic Cryptographic Service API (GCS-API)", Open Group Preliminary Specification, June 1996.

[5] "Microsoft CryptoAPI Application Programmers Guide", Version 1, Microsoft Corporation, 16 August 1996.

[6] "PKCS #11 Cryptographic Token Interface Standard", Version 2.01, RSA Laboratories, 22 December 1997.

[7] "Generic Security Service Application Programming Interface", RFC 2078, John Linn, January 1997.

[8] "DCE Security Programming", Wei Hu, O'Reilly and Associates, 1995.

[9] "SESAME Technology Version 4", December 1995 (newer versions exist but are no longer publicly available).

[10] "Independent Data Unit Protection Generic Security Service Application Program Interface (IDUP-GSS-API)", RFC 2479, Carlisle Adams, December 1998.

[11] "Architecture for Public-key Infrastructure (APKI), Draft 3", The Open Group, 27 March 1998.

[12] "Common Data Security Architecture (CDSA) Version 2.0", The Open Group, May 1999.

[13] "Security Service API: Cryptographic API Recommendation, Updated and Abridged Edition", NSA Cross Organization CAPI Team, 25 July 1997.

[14] "Microsoft Cryptographic Application Programming Interface (CryptoAPI)", Version 2, Microsoft Corporation, 22 December 1998.

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

[16] cryptlib version 3, `http://www.cs.auckland.ac.nz/~pgut001/cryptlib.html`, 1999.

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

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

[19] “Security in Computing”, Charles Pfleeger, Prentice-Hall, 1989.

[20] “Building a High-Performance Programmable, Secure Coprocessor”, Sean Smith and Steve Weingart, Computer Networks and ISDN Systems, **Issue 31** (April 1999), p.831.

[21] “SKIPJACK and KEA Algorithm Specification”, Version 2.0, NSA, 29 May 1998.

[22] “Java Security Architecture”, JDK 1.2, Sun Microsystems Corp, 1997.