KryptoKnight Authentication and Key Distribution System

Refik Molva[1], Gene Tsudik[2], Els Van Herreweghen[2] and Stefano Zatti[2]

[1] EURECOM Institute, Sophia Antipolis, 06560 Valbonne, France
molva@eurecom.fr
[2] IBM Research Laboratory, CH-8803 Rüschlikon, Switzerland
{gts,evh,zat}@zurich.ibm.com

Abstract. This paper describes *KryptoKnight*, an authentication and key distribution system that provides facilities for secure communication in any type of network environment. KryptoKnight was designed with the goal of providing network security services with a high degree of compactness and flexibility. Message compactness of KryptoKnight's protocols allows it to secure communication protocols at any layer, without requiring any major protocol augmentations in order to accommodate security-related information. Moreover, since KryptoKnight avoids the use of bulk encryption it is easily exportable. Owing to its architectural flexibility, KryptoKnight functions at both endpoints of communication can perform different security tasks depending on the particular network configuration. These and other novel features make KryptoKnight an attractive solution for providing security services to existing applications irrespective of the protocol layer, network configuration or communication paradigm.

1 Introduction

The importance of secure communication in today's distributed systems is universally acknowledged. For this reason, much effort has been recently invested into providing security services in a variety of network and operating system environments. One of the best-known efforts is Kerberos [12, 13], a network security service originally developed at MIT and subsequently incorporated into a number of architectures and commercial offerings. In spite of its popularity and widespread acceptance, Kerberos has received its share of criticisms (e.g., [5]). Moreover, it has a number of limitations that preclude its widespread use for all communication paradigms.

In this paper we describe a new network security service called *KryptoKnight*, developed jointly by IBM Zürich and Yorktown Research Laboratories and implemented at the IBM Zürich Research Laboratory. KryptoKnight provides authentication and key distribution services to applications and communicating entities in a network environment. Unlike Kerberos, which uses protocols based on the well-known Needham-Schroeder [9] scheme, KryptoKnight implements a family of novel authentication and key distribution protocols designed with assurance of security with respect to a number of attacks. (The design of the underlying protocols is not treated here; it is addressed in [2], [3] and [4].)

From a user's perspective, KryptoKnight provides facilities and services which are very similar to those of Kerberos. In fact, Kerberos was used both as a stepping stone and a reference point in the design of KryptoKnight. The resulting system offers several advantages over Kerberos:

- a number of novel features of the underlying protocols described in [2], [3] and [4].
- system design that avoids many of the problems attributed to Kerberos that are described in [5].
- support of a major subset of Generic Security Service API (GSS-API) [14].

This paper is organized as follows. An overview of the KryptoKnight architecture is given in the next section. Section 3 describes the protocols in more detail. KryptoKnight's software structure is discussed in Section 4 and Section 6 concludes with a brief summary.

2 KryptoKnight Overview

Three types of principals are involved in KryptoKnight operation: users (or other network entities), programs (services) and authentication servers (ASs). As in Kerberos, the AS is a trusted third-party component which is able to access the principal database, check and create authentication tokens and tickets. The goal of KryptoKnight is to provide authentication and key distribution services to users and programs with the aid of an AS. These services are necessary in order to:

- Support users in delegating their identity to other entities (e.g., application programs),
- Assist communicating entities in mutual authentication by providing the evidence of each entity's delegation by a legitimate user,
- Enable entities to authenticate the origin and contents of exchanged data.

The operating environment is assumed to be untrusted; communication links may be susceptible to wiretapping, interception and replay of messages. Moreover, malicious programs may reside in the same computers as the legitimate ones and may try to access, read and modify their data. It is, therefore, very important for the authentication services not to be vulnerable to masquerading by potential intruders exploiting the exposures inherent to this environment.

KryptoKnight offers four service classes (described in detail below):

1. Single Sign-On (SSO)
2. Two-party Authentication
3. Key Distribution
4. Authentication of origin and contents of data

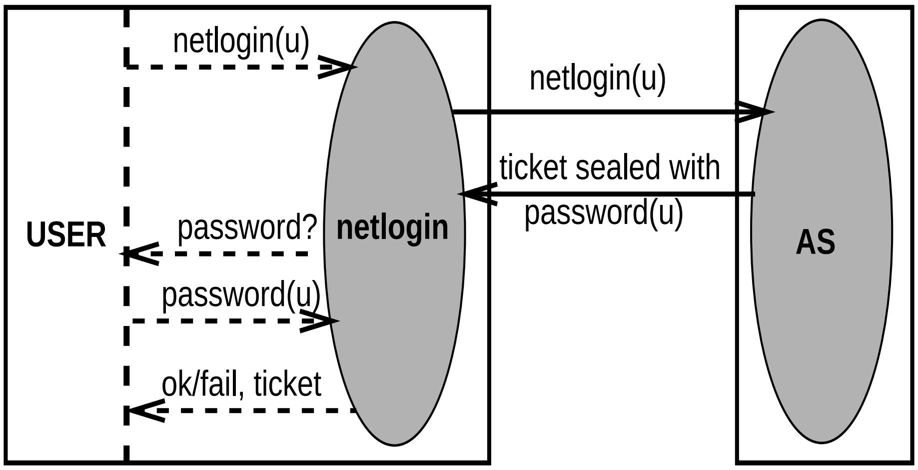

Fig. 1. Single Sign-On Protocol

2.1 Single Sign-On

Prior to starting any activity on his behalf, the user authenticates himself to the KryptoKnight system by executing the *kklogin* command on a public workstation. The purpose of this command is to perform a unified, network-wide login for the user.

Kklogin triggers a message exchange between the local (stub) KryptoKnight program executing the command and the AS (see Figure 1). In the first message, the user tells the AS that he wants to login, specifying his own name. This message allows the AS to authenticate the user as it contains a value which is a function of both the current time and the user's password. This feature is referred to as **pre-authentication**. The second message contains the reply from the AS, which is sealed with a key derived from the user's password. At this point, *kklogin* prompts the user for his password and uses this password to unseal the AS's reply, retrieving the certificate (*ticket*, in Kerberos parlance), contained in it. A successful result implies that the user provided a correct password and has proven his identity. The ticket, obtained by the user, will subsequently be inherited by any activity running on the user's behalf (the format of the ticket will be described in more detail in section 3.2). Using the certificate through KryptoKnight primitives, an entity can communicate with remote peers and prove that it is, in fact, executing on behalf of this particular user. Since user authentication takes place only once (until the user explicitly terminates the login session by executing the *kklogoff* command) and any number of local programs can utilize its result when authenticating themselves to remote programs (services), this operation is called *single sign-on* or SSO.

Alternatively, some programs that reside on relatively trusted systems may need a delegation certificate that lasts longer than the lifetime of a certificate provided through single sign-on. Examples of such programs are application servers or administration programs that enjoy the security of physically protected systems. A long term delegation certificate can be provided through the *kkinstallkey* command executed by the administrator on the system where such a program has to run; in this case, no single sign-on operation is needed.

2.2 Two-Party Authentication

Once delegated by a human user, an entity can use the KryptoKnight Application Programming Interface (API) library to obtain further proofs of its delegation in order to authenticate itself to remote peers. Every entity that engages in authentication is linked with the KryptoKnight API library and, at run-time, issues procedure calls to obtain these services.

An entity, referred to as INITIATOR, starts the authentication process by issuing an API call that returns an authentication message. This authentication message is sent to the remote peer entity, known as RESPONDER, using an peer-entity-specific communication mechanism. When the RESPONDER receives the message, it issues a corresponding API call that verifies the validity of the incoming authentication message. Depending on the authentication parameters, either one-way authentication (where only the INITIATOR is authenticated to the RESPONDER) or mutual authentication (where both parties attain mutual authentication) can be performed. In case of one-way authentication, the first and only authentication message authenticates the INITIATOR to the RESPONDER. If the verification by the RESPONDER is successful, the RESPONDER is sure that the INITIATOR is truly acting on behalf of the user specified in the authentication message. In case of mutual authentication, the exchange of two more authentication messages, created via respective API calls, is required: one will authenticate the RESPONDER to the INITIATOR, the other will complete the protocol by finally authenticating the INITIATOR to the RESPONDER.

2.3 Key Distribution

The authentication protocols described in the previous section require that INITIATOR and RESPONDER share a secret key. In order to obtain a shared secret key, the two parties must first engage in a dialog with a KryptoKnight AS. The specifics of this dialog are discussed in subsequent sections. However, the entire process of contacting the AS, proving one's identity, and receiving keys, is hidden from the entities using KryptoKnight. In other words, whether or not the two parties share a key, they make the same API calls with exactly the same parameters. Once a key is issued and received, it is cached (again, in a transparent manner) for later use.

The relationship between the peer entities and KryptoKnight components involved in program authentication is summarized in Figure 2.

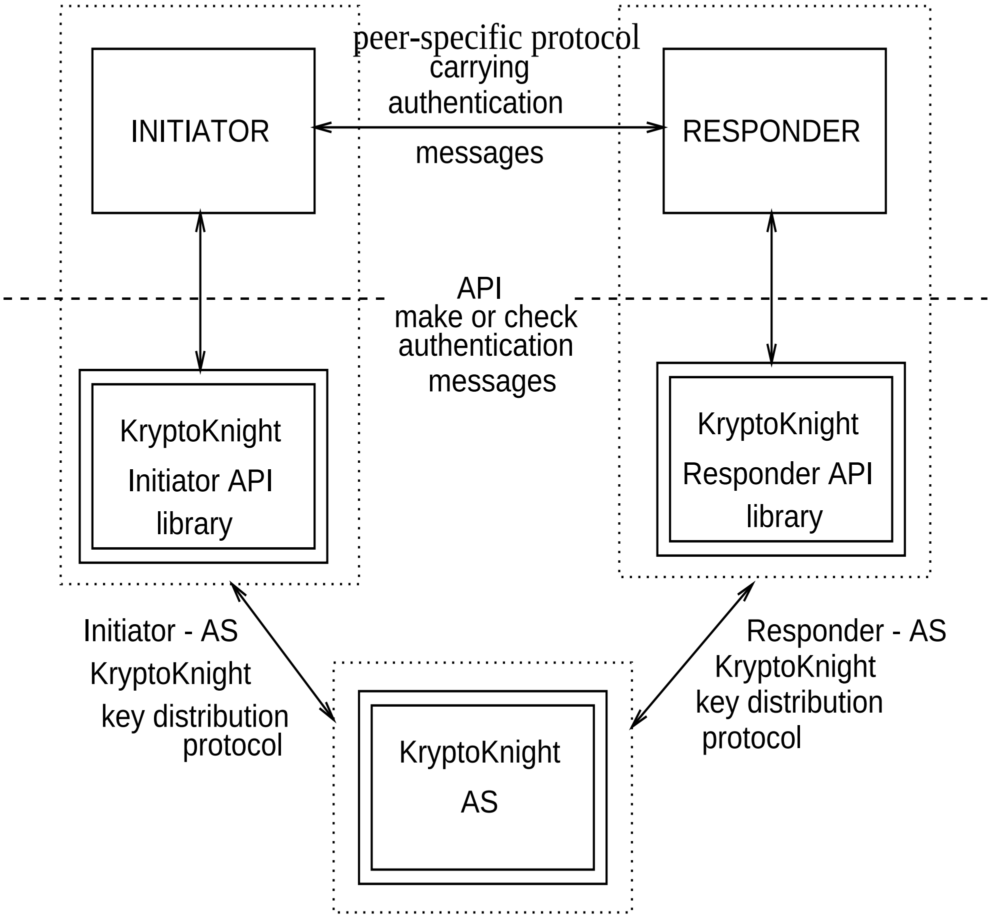

Fig. 2. Program Authentication using KryptoKnight

2.4 Data Origin and Content Authentication

Successful authentication between two entities establishes a KryptoKnight *session*[3] which is characterized by their shared secret key. This session serves as a context for further secure communication between the two parties. It will be terminated either explicitly, by one of the communicating parties sending to the other party a request to end the session, or implicitly upon expiration of the shared key.

Until session termination, the entities can, through the KryptoKnight API, authenticate the contents and the origin of data messages exchanged.

[3] GSS API[14] uses the term *security context*. For the purpose of this discussion the two notions are equivalent.

3 *KryptoKnight* Protocols

The protocols implemented by KryptoKnight perform the following functions:

- *Authentication*: mutual or one-way authentication between the initiator and the responder, between the user and the AS, and between the initiator or the responder and the AS.
- *Key Distribution*: distribution of pair-wise keys to be used for authentication and secure communication between two entities or between an entity and the AS.
- *Integrity Protection*: authentication of contents and origin of application messages.
- *Secure end-of-session*: authenticated session termination.

3.1 Authentication

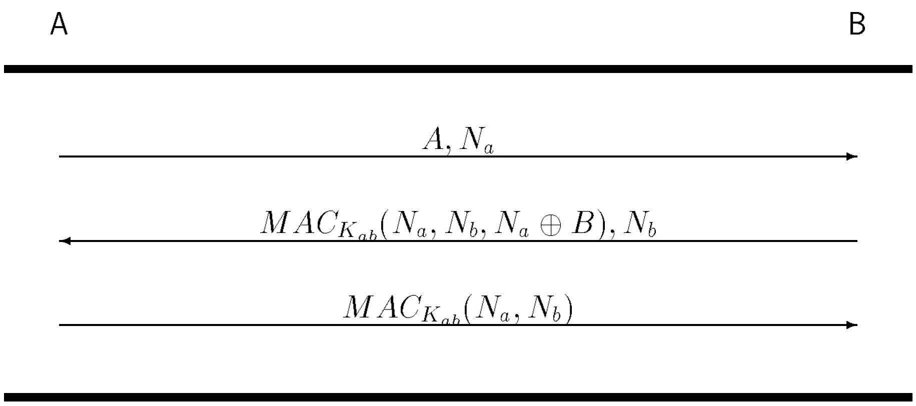

Fig. 3. The Basic Authentication Protocol

The mutual authentication protocol is a three-way exchange between the authenticating parties as depicted in Figure 3. In this figure,

- A and B represent the names of the authenticating parties,
- N_a and N_b are 64 bit nonces generated respectively by A and B,
- K_{ab} is a secret key, shared by A and B,
- $MAC_{K_{ab}}$ is a 64-bit message authentication code, computed over the concatenation of its arguments, using the key K_{ab}. MAC_{ab} can be based on DES, MD4 or MD5 ([10], [11], [7]) (see section 4.6),
- $\oplus$ represents the exclusive-or operation.

The advantages and security features of this protocol are described in [2, 3]. It achieves mutual authentication of two communicating entities.

The one-way authentication protocol is derived from the mutual protocol by A's sending to B the same expression as in the second flow of the latter protocol with N_a replaced by a timestamp, N_b by a nonce generated by A, i.e. N_a, and B's name by the one of A.

3.2 Key distribution

In order to run the authentication protocol, the authenticating parties need to share a secret cryptographic key. Since typically no key is shared other than between the AS and the principals, when a two parties need a shared key, the AS generates a new random key and distributes it to them using the key distribution protocols. Key distribution is performed using a generic message component called a *ticket*. Tickets are used for secure transfer of a secret key from one party called *ticket_issuer* (typically the AS) to another party called *ticket_reader*. A ticket is sealed with a key shared by the ticket_issuer and the ticket_reader. Each ticket is also identified by the name of an entity called *third_party*, which is the party the ticket_reader should share the secret key with. The ticket consists of a set of cleartext fields and an encrypted token. The cleartext fields are:

- Nr: 64 bit nonce generated by ticket_reader
- Ni: 64 bit nonce generated by ticket_issuer; this is actually a keyed one-way function of N_r, e.g., the encryption of N_r under the secret key contained in the ticket
- name of third_party
- the expiration time of the secret key contained in the ticket

The computation of the token (64 bits) is based on an expression similar to the one used in the basic authentication protocol:

$$token = MAC_{K_m}(N_i \oplus third\_party, N_r, N_i \oplus ticket\_issuer, expiration\ time) \oplus K_s$$

where:

- K_m, the *reader_key*, is a key shared by ticket_issuer and ticket_reader;
- K_s, the *session_key*, is the secret key distributed by the ticket (to be used from here on, until expiration of the key, between ticket_reader and third_party).

The name of ticket_issuer and ticket_reader are assumed to be implicitly known by the reader of the ticket, and are therefore not part of the ticket's cleartext fields.

Since knowledge of K_m is necessary to unseal the ticket, the secrecy of K_s is ensured. Moreover, since it is impossible to extract the correct K_s from a ticket that has been tampered with, and since the ticket_reader of a ticket can tell if the extracted K_s value is the right one (by checking whether N_i is actually a cryptographic function of N_r), it is impossible for an intruder to tamper with the ticket without the reader of the ticket being aware.

A ticket can be used for the following purposes:

- to send a good random cryptographic key to a user during the single sign-on
- to send a shared secret key to a pair of entities that need to authenticate one another

Single sign-on tickets Single sign-on is essentially the distribution of a strong key between a user and the AS. During execution of the protocol, the user receives two tickets from the AS, to be read by the user and the AS respectively. The ticket for the user is:

$$\mathrm{ticket(reader = user, issuer = AS, third\_party = AS, reader\_key = K_m, session\_key = K_s)}$$

where:

- K_m is a key derived from the user's password and is stored in the principal database of the AS along with the user's name. KryptoKnight on the user's workstation can generate this key using the user's password and a public password-to-key conversion algorithm.
- K_s is a key that will be shared between the user (reader) and the AS (third_party) for the duration of the single sign-on period (until the user explicitly erases the key by entering the *kklogoff* command). Application programs running on the user's behalf will use this key as the user's personal identification secret in further authentication and key distribution with the AS.

The other ticket is stored by the user and sent to the AS every time the user wants to communicate with the AS. This is a ticket:

$$\mathrm{ticket(reader = AS, issuer = AS, third\_party = user, reader\_key = K_{as}, session\_key = K_s)}$$

where K_{as} is a secret key known only to the AS.

Entity/Peer tickets For key distribution to peer entities, two tickets are made by the AS (one for each entity) both containing the same secret key. For example, the ticket for A to communicate with B is:

$$\mathrm{ticket(reader = A, issuer = AS, third\_party = B, reader\_key = K_a, session\_key = K_{ab})}$$

where:

- K_a is the key distributed to A during the single sign-on or a key installed on both A's workstation and in the AS database if A has a manually installed long-term key (e.g., if A is a server program);
- K_{ab} is the shared key that will be used in the authentication between A and B.

3.3 Integrity protection

Integrity protection and sequencing of messages exchanged between two parties sharing a session key K_s is achieved by sending some *integrity information*, together with the data. This integrity information consists of:

- a session-timestamp, that is a constant for the messages exchanged in the course of a particular session
- a message sequence number, which is incremented by one for each new message within a session
- $MAC_{K_s}(data, session\_timestamp, sequence\_number)$

The MAC expression provides integrity of the data, while timestamp and sequence number are used to detect replayed and out-of-order messages.

3.4 Secure end-of-session

The session established between two peers upon successful completion of an authentication, can be terminated in an authenticated way by either one of the peers by sending to the other a *delete_session* message containing following token:

$$token = MAC_{K_s}(Ni, Nr, Ni \oplus (initiator \mid responder))$$

where (initiator | responder) is the concatenation of the names of both parties. The use of a similar expression as used for authentication ensures it has the same degree of security. However, the different use of the principals' names in the end-of-session token makes it unique and not reusable in any authentication protocol message.

3.5 Authentication Protocol Flexibility

Once peer entities have obtained (either through single sign-on or through the kkinstallkey interface) a secret key corroborating their identity, they can trigger a key distribution and authentication protocol through the API primitives. The actual protocol to be executed between the KryptoKnight system components and between the peer entities is dynamically determined depending on the following conditions

- *authentication type:* the initiator entity can choose between mutual authentication and one-way authentication by using the appropriate initial API primitive.
- *authentication and key distribution sequence:* the protocols several alternatives for interleaving authentication and key distribution flows:
  - **Key distribution prior to authentication:** in the protocol depicted in Figure 4, the KryptoKnight initiator component first exchanges key distribution messages with the AS and, only after this phase is complete, the initiator triggers the authentication exchange with the responder. This scheme potentially enables the AS to enforce access control during

the key distribution phase by granting pair-wise keys based on the authorizations of the initiator and responder. The drawback of this scheme is that the protocol pattern is fixed, that is, the key distribution protocol is always triggered by the initiator and never by the responder, with a lack of adaptability with respect to the existing communication patterns. This disadvantage is obvious in the case of a wide-area network where the AS and the responder are near to one another, while the initiator is very from from the AS.

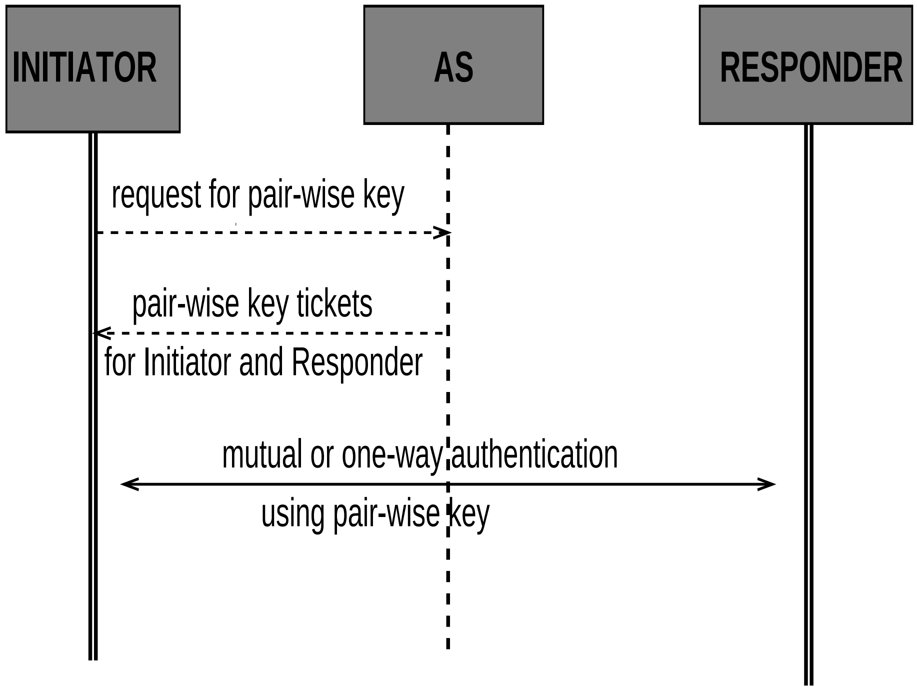

Fig. 4. Key distribution prior to authentication

- **Adaptive protocol (authentication interleaved with key distribution):**[4] in the protocol illustrated in Figure 5, the initiator sends an authentication message to the responder. The further course of the protocol is then determined by the existing connectivity patterns between responder, initiator, and AS. In case the responder can reach the AS, it will exchange key distribution messages with the AS before sending a reply message to the initiator, conveying the new key to the initiator. In case where there is no such connectivity between responder and AS, the responder will send back a reply message to the initiator without having contacted the

[4] The two modes of this protocol are sometimes referred to as *push* and *pull* modes.

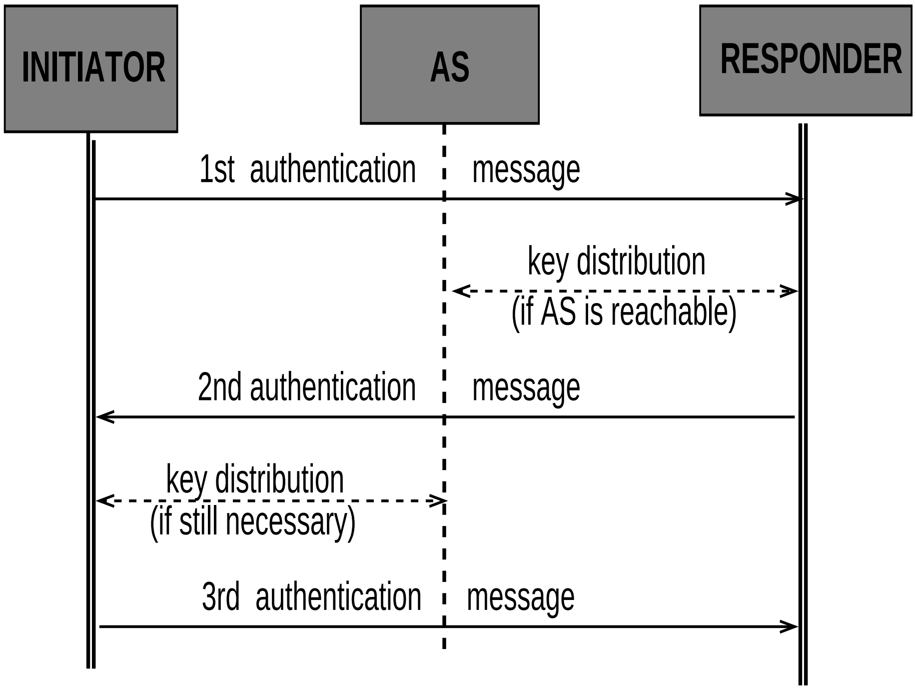

Fig. 5. Adaptive protocol

AS. In this case the initiator will execute the key distribution protocol with the AS and send the key to the responder in the third and final message of the authentication protocol.

The adaptive scheme is applicable only in the case of mutual authentication, because at least a return flow from responder to initiator is necessary to convey the key (possibly even a third flow). The scenario whereby the responder contacts the AS is used by the X9.17 standard [1].

When one-way authentication is performed, the first scheme for key distribution (prior to authentication) is the only one possible, since then the secret key must be conveyed to the responder in the only authentication message available.

The advantage of interleaving the authentication and key distribution flows is that the protocol pattern is not fixed as to whether the initiator or the responder performs the key distribution exchanges with the AS. As a result of this flexibility, the connections between the peer KryptoKnight stubs and the AS can be established in the optimal way in order to minimize the amount of communication and connections between the peer systems and the AS. The KryptoKnight implementation offers the possibility of adapting the protocol execution pattern to a given network by individually setting the authorization to connect to the AS from each node of the network. Using the flexible protocol pattern presented in Figure 5, the KryptoKnight stub on a machine authorized to connect to the AS always exchanges key distribution flows with the AS irrespective of its role as an Initiator or Responder, whereas a stub on a machine without such an authorization always relies on the partner stub for the key distribution.

3.6 Layer Adaptability

As described earlier, KryptoKnight protocol messages consist of a cleartext part (names, nonces, timestamps) and a 64-bit token. The token is a 64-bit integrity check over the message's cleartext part and contains, in the case of a ticket, also the session key contained in the ticket.

Kerberos messages and tickets are much longer than their KryptoKnight counterparts and often contain information that could be retrieved out of the context of the communication. For example, source and destination addresses are already present in most protocol's standard headers. So in most cases it could be sufficient to transmit only an integrity-check over this information without actually transmitting the information itself. But since Kerberos messages and tickets are completely encrypted, they have to be transmitted in their entirety, which imposes a bandwidth and space requirement that cannot be satisfied when authentication is part of a lower-layer protocol (where *vacant* protocol fields are few).

While both Kerberos and KryptoKnight can thus be used for authentication in high-layer application protocols, the compactness of the KryptoKnight messages is clearly an advantage in space-critical, lower-layer protocols.

3.7 Communication Paradigms

Another advantage of KryptoKnight protocols over Kerberos is that in KryptoKnight the initiator and the responder can be peer entities, i.e., they can bear the same privileges with respect to the AS, whereas Kerberos assumes an asymmetric distribution of roles and privileges by distinguishing two types of entities, clients and servers.

In Kerberos (see Figure 6), the initiator is always a client and the responder is always a server. The difference between a client and a server is that the client's secret key (K_c) needed by the AS for sealing the ticket, concisely expressed here as $T\{Kcs\}Kc$, containing the pair-wise key (K_{cs}), is obtained through single sign-on; thus, the AS does not have to store K_c in its database since it can retrieve it from $T\{Kc\}Kas$ that was distributed during single sign-on. On the other hand the server's key (K_s), used to seal the pair-wise key (K_{cs}) in $T\{Kcs\}Ks$, must be a long-term key which the AS retrieves from its principal database, since the Kerberos protocol flows do not permit the server to provide the AS with any information prior to the distribution of the pair-wise key. Consequently, the AS must store the secret key of each server (K_s), whereas it does not have to remember the secret keys of the clients that can be dynamically retrieved from

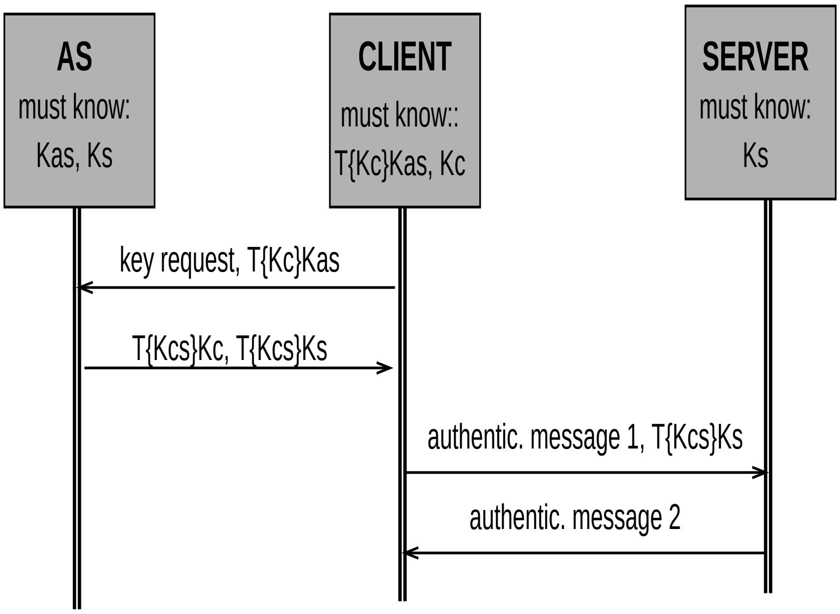

Fig. 6. Asymmetric Client-Server requirements in Kerberos

single sign-on tickets. This asymmetry has a very strong implication on the users of the Kerberos protocol:

– any entity that can play the role of a responder in the authentication process must be configured as a server, that is, a long-term key that is equivalent to the one obtained through single sign-on on the client systems must be manually installed on each server system and entered into the principal database of the AS.
– even if the entities playing the role of initiator and responder are genuinely peers, in order to comply with the asymmetry imposed by the Kerberos authentication protocol they must be configured as a client and a server in the above described sense.

In KryptoKnight, no particular asymmetry is imposed by the protocol, the initiator and the responder can be configured either both as clients, both as servers or one as a client and the other as a server. This property follows from the adaptive KryptoKnight protocol scheme (Figure 5) where there are sufficient flows exchanged between the initiator and the responder before the key distribution (triggered either by the initiator or by the responder) so that the key distribution request sent to the AS can always contain single sign-on tickets for both parties. Only the particular KryptoKnight protocol where key distribution takes place before any authentication exchange like in Kerberos (as shown in Figure 4) suffers from the same shortcoming as the Kerberos protocol, requiring the responder to be a server.

3.8 Nonces vs Timestamps

In MIT Kerberos the problem of replay by wiretapping intruders is solved by using timestamps that prove the freshness of messages. This technique on one hand largely simplifies the protocol design, but has on the other hand some major drawbacks:

- authenticating parties must have access to synchronized and reliable clocks in order to get timestamps and to verify them.
- the strength of the authentication is inversely proportional to the skew between the clocks; in other terms, the tighter the synchronization, the stronger the security.
- clock servers that seem to be a reasonable solution for several applications do not suit the requirement of authentication because an authentication service requires a trusted clock server and a trusted clock server must be authenticated prior to its use.

In KryptoKnight, the basic authentication protocol (Figure 3) as well as the three-party (Figure 4 and 5) and single sign-on protocols use nonces for replay detection. Nonce-based authentication requires at least one more flow than the timestamp-based one, but avoids the problems of relying on an external mechanism like synchronized clocks. Nonetheless, one-way authentication in KryptoKnight has to be based on timestamps, being designed to be compatible with the one-way Kerberos protocol that uses only one message.

3.9 Exportability

As described in [2, 3, 4], the underlying authentication protocols do not require a full-blown encryption system, but, rather a strong one-way hash function. As designed and implemented, KryptoKnight is freely-exportable as it does not make use of data encryption. Both MD4 and MD5 are one-way hash functions not covered by export regulations. Even with DES, KryptoKnight uses only the encryption function (i.e., the code does not include DES decryption.) Where secrecy is absolutely necessary, e.g., in hiding session keys during key distribution or protecting principals' key in the database, a simple XOR-ing technique is used (see [2, 3]).

4 Software Features

The KryptoKnight software consists of separate modules organized in three layers (see Figure 7):

- The *protocol entity layer* consists of modules implementing the active entities for the different KryptoKnight protocols, including the user, the initiator, the responder, the AS, and the admin modules.
  The KryptoKnight API consist of *initiator* verbs, *responder* verbs and *common* verbs (used by both initiator and responder) and can be called either

directly by the peer entities or indirectly through the GSS API interface verbs, that have been implemented on top of it.
- the *basic operations layer* implementing basic operations used by the protocol entities like the computation of security-critical fields by the token module, the management of the key cache, the network interface, etc.
- the *cryptographic layer* that includes the encryption or checksum function and an optional key storage facility that should exist only if a cryptographic device is available.

The advantage of this modular implementation is that any individual module can be replaced or modified without any impact on the remaining modules. In the remaining sections the most interesting modules will be addressed.

4.1 The API Module

The API Module implements the verbs enabling entities using KryptoKnight (e.g., application programs) to authenticate to one another (*initiator* and *responder verbs*), and subsequently exchange integrity-protected data and securely terminating the session (*common*).

4.2 The User Module

The User module implements the user commands *kklogin, kklogoff, kkinstallkey, kkpasswd, kkservrekey.*

As discussed in section 2.1, *kklogin* performs single sign-on for users. The resulting ticket can be used by entities running on behalf of the user to authenticate to other entities, and is destroyed when the user performs a *kklogoff.*

kkinstallkey installs a long-term masterkey for a server program. A user can also change his password by executing the *kkpasswd* command. Like kklogin, this command triggers a message exchange between the local kkpasswd program and the AS, during which the user is prompted to enter both the old and the new password. If the AS is able to verify the identity of the user and the integrity of the message containing the user's new password, it will replace the user's password in its database by the new one.

Long-term capabilities can be renewed using the *kkservrekey* command.

4.3 The Admin Module

The administrative interface was taken from Kerberos 5. Using the Admin commands a human user acting as administrator can access the Principal Database and manipulate the principal definitions. All operations performed by the Admin module are local operations on the site where the Principal Database resides.

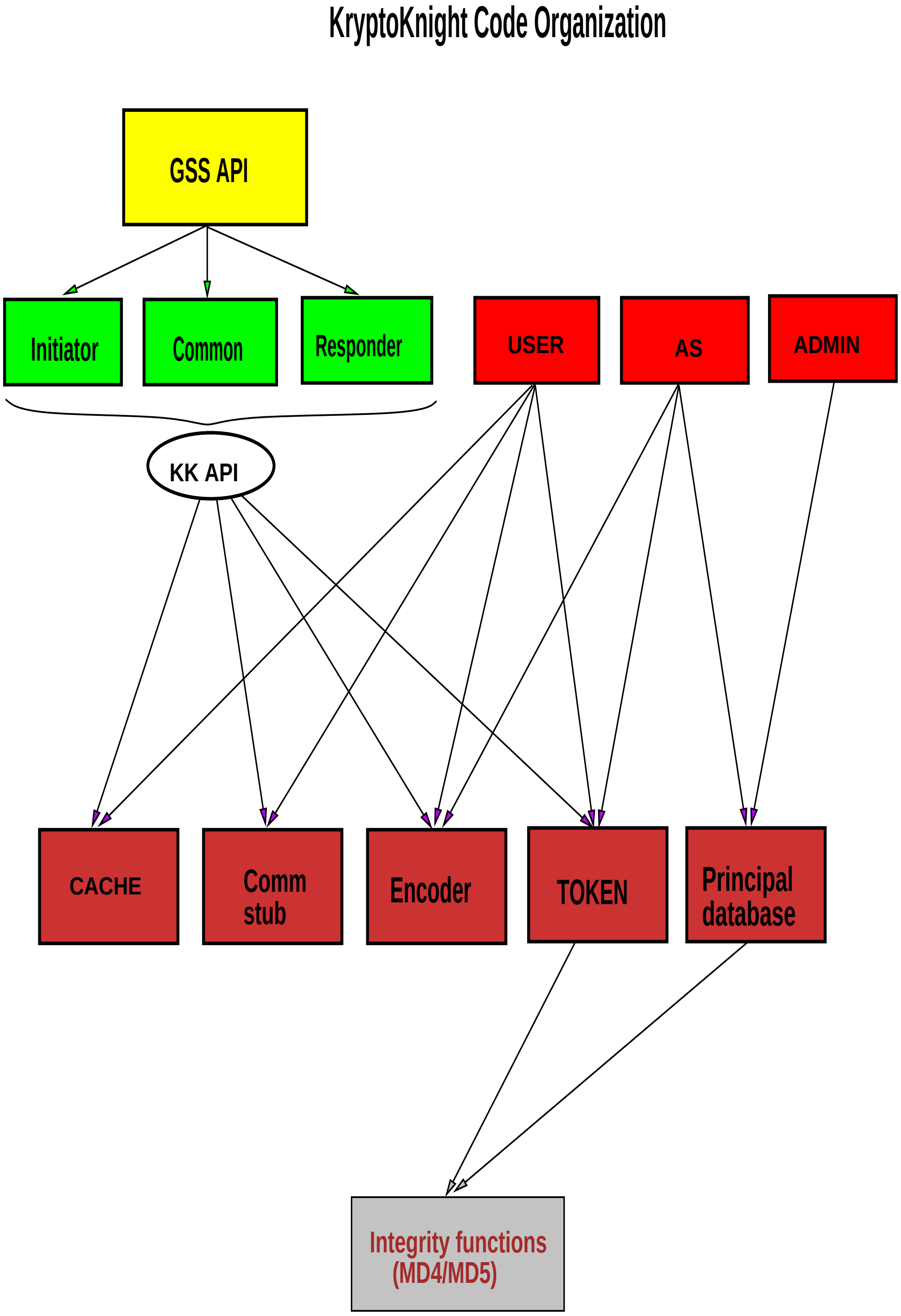

Fig. 7. KryptoKnight software structure

4.4 The Token Module

The token module implements the algorithms for computing the cryptographic token included in tickets, authentication and end-of-session messages. The computation of protocol-specific expressions is confined to this module and all the active protocol entities involved in KryptoKnight protocols use the token module to make and verify security-critical messages. The generation and the verification of all tickets and authenticators is performed through a set of primitives that constitute the token interface.

A feature of the token module is the possibility of isolating secret keys into a single module within the software. If safe storage is available to store a master key-encryption key for each installation of KryptoKnight, e.g., if a cryptographic hardware device is used or if a strictly local file exists (like /tmp on some UNIX implementations), then the token interface can provide a clear boundary beyond which no secret key appears in cleartext. In this case the keys exchanged at the token module interface are encrypted by the token module under the master key-encryption key and never appear in cleartext except inside the token module when they are included in or extracted from a ticket. The main advantage of this confinement is that it eliminates any exposure to wiretapping (on LANs) while session keys and tickets are manipulated through a network file system protocol (like SUN NFS) operating on the cache files where higher-level modules store them.

4.5 Communication Stub

The communication stub has the sole task of handling all communication with the AS(s). Every workstation or host that runs *kryptonized* entities (including single sign-on) must have a running communication stub. Of course, its services are used only when AS involvement is necessary, i.e., for single sign-on and key distribution. Communication stub is not used for two-party authentication which, as described above, is obtained by exchanging entity-specific messages.

KryptoKnight interposes a unified process, called *the communication stub*, between all *kryptonized* entities running on the same workstation and the AS(s). However, the entities are completely **unaware** of any communication with the AS. Of course, this means that the presence of the communication stub is entirely transparent. Whenever there is a need to contact the AS, an AS-bound message is composed and the AS interface function (invisible to the API), *send_to_AS* is invoked. This function, in turn, uses an IPC mechanism (e.g., a UNIX pipe) to deliver the message to the communication stub process. When the AS replies, the communication stub forwards the reply (again, via IPC) back to the caller.

Another benefit of having a unified AS interface is the ease of changing communication platforms. In its present incarnation, KryptoKnight assumes TCP/IP- or SNA-based communication. (In other words, there are two distinct communication stubs.) However, if other, e.g., NETBIOS or X.25, communication platforms are to be supported in the future, the conversion task is limited to re-writing a fairly short ($\leq$ 100 lines) and trivial piece of code.

4.6 The Integrity/Encryption Module

but The Integrity/Encryption Module provides integrity (for the Message Authentication Codes and for integrity-protection of application messages) and secrecy functions (for protection of the principals' keys in the database) to the token and PDB (Principal Database) modules.

Currently several implementations of this module exist. One is based on DES [7] used in Cipher Block Chaining (CBC) mode. The other two are based on one-way hash functions: MD4 and MD5 [10, 11]. Both MD4 and MD5 had to be slightly augmented to support the use of a key as a secret secret and/or suffix as described in [16]). Secrecy is achieved, as described in Section 3.9 above, with a simple XOR-ing technique.

5 Future Considerations

This section discusses some issues and items for future work.

5.1 Separation of AS and Database Master Keys

Currently, the database master key (with which the principal's keys in the database are encrypted) and AS ticket key (K_{as}) are one and the same. This is an artifact of Kerberos [13] and it should be changed as there is no intrinsic requirement for these two keys to be the same. This issue must, of course, be addressed in the course of separating AS and PDB functions (e.g., AS accesses PDB via RPC).

5.2 Encryption/Signature Methods and Key Length

All encryption/integrity operations in KryptoKnight are currently carried out using a single method: MD4/MD5 with secret prefix and/or suffix [16] or DES [7]. This is not an inherent requirement. There are three distinct tasks requiring encryption and/or integrity services:

1. Key distribution to principals via ticket tokens. When a principal requests a single sign-on ticket or an application ticket AS distributes the resulting SSO- or session-key as part of a ticket token.
2. Key distribution to AS (by AS). The single sign-on ticket to be read by the AS contains the same SSO-key as the ticket for the principal, however, the token is computed using the AS's own ticket key K_{as}.
3. Encrypting principals' keys in the database (PDB). This service is used by the AS and the PDB manipulation programs (Admin) to hide the contents of principals' keys.

It is not necessary for these three tasks to use the same encryption/integrity mechanism. AS, for example, could take advantage of stronger encryption methods such as RSA [15]. This should be trivial to implement since the AS would be the only entity using it. The same is true of Item 3.

All keys in the current release, including the AS ticket key and the PDB master key, are 64 bits long. For reasons alluded to above, session and SSO keys must remain 64 bits since their size is tied to the size of tokens. However, there is no such requirement for either the AS ticket key or the PDB master key. Both can be longer, e.g., 128 or 512 bits.

5.3 Inter-Domain Support

KryptoKnight currently has no provisions for supporting inter-domain authentication and access control. However, a number of inter-domain authentication scenarios have already been designed. They are the natural extensions of the current intra-domain protocol variations to the inter-domain case. Future work is likely to include the support for the inter-domain environment.

6 Summary

This paper presented KryptoKnight, an authentication and key distribution system which provides security services to applications and communicating entities in a network environment. KryptoKnight is a fully functional system; it is implemented (and operational) on both IBM RS/6000 and IBM PS/2 machines under AIX operating system. However, there are no inherent dependencies on either current hardware or operating system platforms.

In summary, KryptoKnight was designed with the benefit of Kerberos' experience and with the goal of providing a high degree of compactness and flexibility. Its message compactness, protocol flexibility and exportability make KryptoKnight an attractive solution for securing existing applications and communicating systems at any protocol layer, irrespective of network configuration or communication paradigm.

7 Acknowledgements

We are grateful to Phil Janson and Liba Svobodova for their careful readings and insightful comments on the drafts of this paper. We also thank the anonymous referees for their helpful reviews.

References

1. ANSI *Banking - Key Management (Wholesale)*, ISO 8732 / ANSI X9.17, 1988,.
2. R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, M. Yung, *Systematic Design of Two-Party Authentication Protocols*, Proceedings of Crypto'91, August 1991.
3. R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, M. Yung, *Systematic Design of a Family of Attack-Resistant Authentication Protocols*, IEEE JSAC Special Issue on Secure Communications, to appear in 1993.

4. R. Bird, I. Gopal, A. Herzberg, P. Janson, S. Kutten, R. Molva, M. Yung, *A Modular Family of Secure Protocols for Authentication and Key Distribution* Draft, in submission to IEEE/ACM Transactions on Networking, August 1992.
5. S.M. Bellovin, M. Merritt, *Limitations of the Kerberos Authentication System*, ACM SIGCOMM Computer Communication Review, October 1990.
6. W. Diffie and M. Hellman, *New Directions in Cryptography,* IEEE Transactions on Information Theory, November 1976.
7. National Bureau of Standards, *Federal Information Processing Standards,* National Bureau of Standards, Publication 46, 1977.
8. T. Lomas, L. Gong, J. Saltzer, R. Needham, *Reducing Risks from Poorly Chosen Keys*, Proceedings of ACM Symposium on Operating System Principles, 1989.
9. R. Needham and M. Schroeder, *Using Encryption for Authentication in Large Networks of Computers*, Communications of the ACM, December 1978.
10. R. Rivest, *The MD4 Message Digest Algorithm*, Proceedings of CRYPTO'90, August 1990.
11. R. Rivest, *The MD5 Message Digest Algorithm*, Internet DRAFT, July 1991.
12. J. Steiner, *The Kerberos Network Authentication Service Overview*, MIT Project Athena RFC, Draft 1, April 1989.
13. J. Steiner, C. Neuman, J. Schiller, *Kerberos: An Authentication Service for Open Network Systems*, Proceedings of USENIX Winter Conference, February 1988.
14. J. Linn, *Generic Security Service Application Program Interface*, Internet Draft, Jun1 1991.
15. R. Rivest, A. Shamir and L. Adleman, *A Method for Obtaining Digital Signatures and Public Key Cryptosystems*, Communications of the ACM, February 1978.
16. G. Tsudik, *Message Authentication with One-Way Hash Functions*, Proceedings of IEEE INFOCOM 1992. May 1992.

This article was processed using the LaTeX macro package with LLNCS style