Design, Verification and Implementation of an Authentication Protocol*

Thomas Y.C. Woo† Simon S. Lam
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

Abstract

We present an account of the entire development cycle (i.e., design, specification and verification, and implementation) of a realistic authentication protocol, which is part of a security architecture proposed by us. The protocol's design follows a stepwise refinement process, which we illustrate. Our account of its specification and verification provides a practical demonstration of a proposed formal analysis approach. For its implementation, we adopt the recently proposed GSS-API standard. We describe the mapping from our protocol to GSS-API, which can serve as a reference for other protocol implementations. We believe that the global perspective presented in this paper would be of great value to protocol designers, verifiers, and implementors, and contribute toward bridging the gap between the theory and practice of authentication protocol design.

1 Introduction

Authentication is a fundamental concern in the design of secure distributed systems [10, 21]. For example, in the prevalent client-server computing paradigm, a server must verify a client's identity before it can make authorization decisions; similarly, a client must ascertain a server's legitimacy before it would proceed with its service request. In a distributed environment, authentication is typically carried out by protocols, called *authentication protocols*. The primary goal of an authentication protocol is to establish the identities of principals that participate in the protocol. Typical principals include users, workstations, processes, and so on. Many authentication protocols, however, also accomplish a secondary goal, namely, the distribution of a new secret session key for future private communication among the participating principals.

The design of authentication protocols is notoriously error-prone. Indeed, many authentication protocols have been published and later found to contain subtle weaknesses or flaws [4, 13]. Two factors contribute to this: (i) the lack of well-established guiding principles for authentication protocol design; and (ii) the use of informal operational reasoning in protocol analysis.

Recently, much research has been directed toward remedying these two problems. To address (i), basic design principles corresponding to symmetric and asymmetric cryptosystems are discussed in [21], and the design of a family of authentication protocols is systematically demonstrated in [3]. To address (ii), a number of formal approaches have been proposed specifically for the analysis of authentication protocols [4, 6, 9, 23]. These research efforts represent useful steps toward understanding how to design and analyze authentication protocols.

However, there is still a significant lag between the development of formal methods and their practical application. Part of the reason can be attributed to the relative immaturity of some of the proposed methods. But a more important reason, we believe, is the lack of practical and systematically worked-out examples that protocol designers and implementors can study and use as a reference in their work. Most expositions of analysis techniques are illustrated with toy examples, specified in a highly abstract notation, with little guidance as to how the techniques can be applied to more realistic protocols. It is rare, if at all, to see a presentation of the entire development cycle (i.e., design, specification and verification, and implementation) of an authentication protocol. We believe that such a complete account is crucial in bridging the gap between theory and practice, and would be of great value to practitioners. Moreover, it can serve as an empirical demonstration of the practical strength of a proposed formal approach, and the protocol will be useful as a benchmark for comparing different formal approaches.

*Research supported in part by NSA INFOSEC University Research Program under contract no. MDA 904-91-C7046 and MDA 904-93-C4089, and in part by National Science Foundation grant no. NCR-9004464.

†The current affiliation of Thomas Woo is Wireless Networking Research Department, AT&T Bell Laboratories.

0-8186-6685-4/94 $4.00 © 1994 IEEE

Addressing this need is an important goal of this paper. Specifically, we take a realistic peer-to-peer authentication protocol that is a key component of a security architecture (proposed by us) together with an analysis approach we have chosen, and walk the reader through the protocol's design, specification and verification, and implementation phases. We explore the main issues, and their relationships, encountered in each of these phases. Our work is in the same spirit as a number of previous studies where a realistic system is formally specified and verified, except that our study is for a different problem domain [5, 8].

Additionally, this paper has several other contributions. First, it presents a stepwise refinement process for deriving authentication protocols. The process, albeit informal, is useful in developing new protocols. Second, the authentication protocol presented in this paper is interesting in its own right. It overcomes many of the drawbacks found in existing client-server authentication protocols (e.g., Kerberos [18], SPX [20]). It is based on the use of both symmetric and asymmetric cryptosystems and can be made *adaptive* in systems with a changing communication topology. The protocol can be viewed as a simple extension of the three-way handshake protocol used in the connection establishment phase of TCP [7], and hence can be readily adapted to provide transport level authentication. Third, it provides a detailed and practical demonstration of the application of a particular analysis approach. Fourth, we adopt the recently proposed *Generic Security Service Application Program Interface* (GSS-API) draft standard [11] for our protocol implementation. We describe the mapping from our protocol to GSS-API, which can serve as a valuable reference for other implementations.

The balance of this paper is organized as follows. In the next three sections, we present, in order, the design, specification and verification, and implementation of the authentication protocol mentioned above. In Section 5, we discuss various issues encountered in different phases of the development cycle and suggest topics for further research. Section 6 is our conclusion. Due to length limitation, we omit the detailed proofs in this paper.

2 Protocol Design

The protocol in this paper is a peer-to-peer authentication protocol. It is similar in functionality and structure to most existing authentication protocols (e.g., Kerberos [18], SPX [20]). Specifically, it mutually authenticates two communicating peers with the help of a trusted third-party server. In this paper, we consider only *intradomain* authentication; that is, we assume that all principals are under a single authority and trust a common server. For *interdomain* authentication, additional mechanisms (e.g., certificate hierarchy navigation) are needed, which are beyond the scope of this paper.

Our protocol design follows a stepwise refinement process. We start with a very simple protocol that is intuitively correct. Then we successively analyze and refine it to relax certain assumptions. We try to make small changes in each refinement step, so that we can informally observe that the desirable security properties are preserved. Most of our refinements steps are based on known design principles [1]. We caution however that such a refinement process is informal, and does not obviate a formal analysis of the protocol at the end. On the other hand, a protocol obtained from such a refinement process, using known design principles, is more likely to be correct and easier to be proved correct. A stepwise derivation procedure has previously been used in [3]; their focus is on a much lower level of detail (e.g., message content) while we focus more on protocol steps.

A key idea in our design is separation of key distribution and mutual authentication functions. That is, we partition the protocol design task into two smaller components: (1) the design of a protocol that distributes a new session key to two principals; (2) the design of a protocol that mutually authenticates two principals using the newly distributed session key. The final protocol is obtained from a "composition" of these two protocols.

We made a number of other design decisions. First, the protocol should not be based solely on the use of symmetric cryptosystems. Asymmetric cryptosystems should be used for control functions (e.g., key distribution), whereas symmetric cryptosystems are preferred for key handshake and data transfer. This is partly because asymmetric cryptosystems generally allow easier key management and require less "trust", and partly because the rest of our security architecture (e.g., the login protocol) makes use of asymmetric cryptosystems. Second, the protocol should avoid the use of timestamps; nonces should be used exclusively instead. (This eliminates the need for synchronized clocks.) Third, the protocol should be symmetric with respect to the participating principals. In other words, the processing requirements of both peers are similar; hence their roles can be easily interchanged. This is unlike most existing protocols, e.g., Kerberos, where client and server processing requirements are very different.

Notation. Principals are denoted by upper case letters, e.g., P, Q. The shared key between P and Q is denoted by k_{PQ}. The public and private keys of P are denoted respectively by k_P and k_P^{-1}. The concatenation of messages m and m' is denoted by m, m'. Encryption of a message m by a key k is denoted by $\{m\}_k$. A protocol is presented as a sequence of *protocol steps*. Each protocol step is written in the form "$P \rightarrow Q : m$" which represents

the communication of message m from P to Q; or in the form "P : *action*" which represents an internal action of P. □

2.1 Key Distribution

We begin with the following simple distribution protocol, to be referred as Π_1:

$$\begin{array}{lll} (1) & S & : \text{ generate secret } s \\ (2) & S \rightarrow P & : \{\{P,T,s\}_{k_S^{-1}}\}_{k_P} \end{array}$$

S is a server (a fixed principal) whose sole function is to generate secrets and distribute them to various principals in the system. T is a timestamp.

The correctness of Π_1 can be informally argued as follows: Assuming synchronized clocks, T provides a *timeliness* guarantee. The inner encryption by S's private key certifies that s does indeed come from S (i.e., *origin authenticity*), assuming that S's public key is known by all principals in the system. The outer encryption by P's public key ensures the *secrecy* of s. Lastly, the inclusion of P's name binds P with s, thus making explicit the intended receiver of s.

The problem of key distribution is simply a specialized form of secret distribution, in which a secret (i.e., a new session key) is distributed to a pair of principals. Thus, Π_1 can be refined to the following key distribution protocol, Π_2:

$$\begin{array}{lll} (1) & S & : \text{ generate key } k \\ (2a) & S \rightarrow P & : \{\{P,Q,T,k\}_{k_S^{-1}}\}_{k_P} \\ (2b) & S \rightarrow Q & : \{\{P,Q,T,k\}_{k_S^{-1}}\}_{k_Q} \end{array}$$

The second and third lines above are labeled as (2a) and (2b) to reflect that the order of their executions is unimportant. Note that each key k is generated for a specific pair of principals.

Π_2 is unsatisfactory in two ways: (1) The distribution is initiated by S, instead of P or Q. Thus, P and Q may to have wait indefinitely before communication can begin. (2) The use of a timestamp requires synchronization of clocks. We fix them in Π_3 below. Specifically, a *challenge-response* step with a nonce is used in place of timestamps. Note that each principal generates its own nonce.

$$\begin{array}{lll} (1a) & P & : \text{ generate nonce } n_P \\ (1b) & Q & : \text{ generate nonce } n_Q \\ (2a) & P \rightarrow S & : P, n_P, Q \\ (2b) & Q \rightarrow S & : Q, n_Q, P \\ (3) & S & : \text{ generate key } k \\ (4a) & S \rightarrow P & : \{\{P,Q,n_P,k\}_{k_S^{-1}}\}_{k_P} \\ (4b) & S \rightarrow Q & : \{\{P,Q,n_Q,k\}_{k_S^{-1}}\}_{k_Q} \end{array}$$

Although Π_3 allows either principal to initiate, it does not eliminate the problem entirely. Specifically, the requests from P and Q are asynchronous, and S responds only after it has received both requests, thus it is possible for either party to wait indefinitely. We can easily remedy this by having the request from one principal forwarded through the other principal, as shown in Π_K below:

$$\begin{array}{lll} (K1) & P & : \text{ generate nonce } n_P \\ (K2) & P \rightarrow Q & : P, n_P \\ (K3) & Q & : \text{ generate nonce } n_Q \\ (K4) & Q \rightarrow S & : P, n_P, Q, n_Q \\ (K5) & S & : \text{ generate key } k \\ (K6) & S \rightarrow Q & : \{\{P,n_P,Q,n_Q,k\}_{k_S^{-1}}\}_{k_Q} \\ (K7) & Q \rightarrow P & : \{\{P,n_P,Q,n_Q,k\}_{k_S^{-1}}\}_{k_P} \end{array}$$

Strictly speaking, line (K7) above is not a forwarding step. Q has to decrypt the message it receives in line (K6) and re-encrypt the result with P's public key before it can send out the message in line (K7). This is, however, "equivalent" to a simple forwarding if P's public key is consistently known by both S and Q.

2.2 Mutual Authentication

The key distribution protocol in the last subsection establishes a new secret session key between two principals. This key can be used to mutually authenticate the two principals.[1] We show below a fairly straightforward two-party mutual authentication protocol; we refer to it as Π_M. Note that Π_M assumes that a key k is already shared between the principals.

$$\begin{array}{lll} (M1) & P & : \text{ generate nonce } n_P \\ (M2) & P \rightarrow Q & : P, n_P \\ (M3) & Q & : \text{ generate nonce } n_Q \\ (M4) & Q \rightarrow P & : \{n_P, n_Q\}_k \\ (M5) & P \rightarrow Q & : \{n_Q\}_k \end{array}$$

It might appear that Π_M is susceptible to *interleaving* attacks [3]. This is, however, not the case because we assume that k is distributed afresh each time just prior to an authentication exchange. Thus, unlike the typical setting of a usual two-party mutual authentication, authentication is never performed again using the same key k.

A number of other two-party mutual authentication protocols have been proposed in the literature [3]. Several of them fit our requirements; we prefer the above for its simplicity. In any case, with our compositional design approach, it is relatively easy to substitute another protocol in place of Π_M. (See Section 5 for more discussion.)

[1] This step is sometimes referred to as *key handshake*.

(KM1)	P	:	generate nonce n_P
(KM2)	$P \rightarrow Q$	:	P, n_P
(KM3)	Q	:	generate nonce n_Q
(KM4)	$Q \rightarrow S$	:	P, n_P, Q, n_Q
(KM5)	S	:	generate key k
(KM6)	$S \rightarrow Q$	:	$\{\{P, n_P, Q, n_Q, k\}_{k_S^{-1}}\}_{k_Q}$
(KM7)	$Q \rightarrow P$	:	$\{\{P, n_P, Q, n_Q, k\}_{k_S^{-1}}\}_{k_P}$, $\{n_P, n_Q\}_k$
(KM8)	$P \rightarrow Q$	:	$\{n_Q\}_k$

Figure 1: Protocol Π_{KM}

2.3 Combined Protocol

We obtain the final protocol by "composing" the two subprotocols Π_K and Π_M. The resulting protocol, denoted by Π_{KM}, is shown in Figure 1.

The "composition" is carried out as follows: The nonces n_P and n_Q generated in Π_K can be reused in Π_M. Thus, steps (M1) to (M3) can be combined with steps (K1) to (K3) in Π_K, and hence eliminated. The message sent in step (M4) of Π_M can be piggybacked onto the message in step (K7) of Π_K. In the sequel, we will refer to P as the *initiator* and Q the *responder*.

Besides satisfying the design requirements laid out above, protocol Π_{KM} is interesting in another regard: it can be viewed as a "secure" extension of the ordinary three-way handshake used in TCP connection establishment. Specifically, steps (KM2), (KM7) and (KM8) correspond to the three steps of three-way handshake. In (KM2), the initiator communicates its sequence number (nonce n_P) to the responder. In (KM7), the responder acknowledges the initiator's sequence number as well as forwarding its own (nonce n_Q). Finally, the initiator acknowledges the responder's sequence number in (KM8). The encryption required for (KM2), (KM7) and (KM8) together with the extra messages to S represent the "cost" of adding security to three-way handshake.

3 Protocol Specification and Verification

There are three tasks in verifying a protocol: (i) Specify the protocol in a notation that has a precisely defined semantics. (ii) Formalize the high level goals of the protocol, e.g., as correctness properties in the form of *assertions* with well-defined semantics. (iii) Demonstrate that the assertions of correctness properties are satisfied by the protocol specification based upon a well-defined notion of *satisfaction*.

In this paper, we employ the verification methodology proposed by Woo and Lam [23, 24], which was designed at a level of abstraction relatively close to that of protocol implementation. In other words, the semantic gap between the formal protocol being verified and the protocol being implemented is small and can be intuitively justified (as opposed to most logical approaches such as BAN logic whose *idealization* step may introduce a large semantic gap).

In the next subsection, we provide a brief overview of Woo and Lam's methodology. Then in the following subsections, we present, in order, a formal specification of Π_{KM} in the notation of [23], a correctness specification and a summary of the proof that the correctness specification is satisfied by the formal protocol specification.

3.1 Methodology Overview

Typical high-level goals of an authentication protocol are the following:

- *Authentication* — For each participating principal, upon successful termination of its protocol execution, it should be assured that it is "talking" to the principal it has in mind.
- *Key Distribution* — The new session key distributed should at most be known by the principals it is intended for.

In Woo and Lam's methodology, these goals are formalized using two types of correctness properties, namely, *correspondence* and *secrecy*. Informally, correspondence specifies that different principals in an authentication protocol execute the protocol in a locked-step fashion. In particular, when an authenticating principal finishes its part of the protocol, the authenticated principal must have been present and participated in its part of the protocol. Correspondence addresses the authentication goal. Secrecy specifies that certain information (e.g., private keys, new session keys) should not be accessible to an intruder. Secrecy addresses the key distribution goal.

Correspondence and secrecy are two distinct properties, although their verification may be inter-dependent. Correspondence properties are formalized using *correspondence assertions*. A correspondence assertion specifies that certain transitions must be related in a one-to-one fashion. Secrecy properties consist of the *general secrecy condition* (GSC) which specifies that a session key being distributed in an ongoing exchange cannot be discovered by an intruder; and *secrecy assertions*, which specify that certain message terms are not accessible to an intruder.

A correspondence assertion is typically proved by applying a set of *proof rules*; while a secrecy assertion is proved using standard invariance verification techniques.

Initial Conditions:

(IC1) $\forall x, y : x$ has $y \wedge x$ has k_y
(IC2) $\forall x, y : x$ has $k_y^{-1} \Leftrightarrow x = y$
(IC3) Z has $\mathbf{X} \Leftrightarrow [\mathbf{X} \in$ SYS
$\vee \exists x : \mathbf{X} = k_x \vee \mathbf{X} = k_Z^{-1}]$

Initiator (i) Protocol:

(I1) BeginInit (r)
(I2) NewNonce (n)
(I3) Send ($r, [i, n]$)
(I4) Receive ($r, [\{\{i, n, r, \mathbf{N}, \mathbf{K}\}_{k_S^{-1}}\}_{k_i}, \{n, \mathbf{N}\}_{\mathbf{K}}]$)
(I5) Send ($r, [\{\mathbf{N}\}_{\mathbf{K}}]$)
(I6) Accept ($\mathbf{K}$)
(I7) EndInit (r)

Responder (r) Protocol:

(R1) BeginRespond (i)
(R2) Receive ($i, [i, \mathbf{N}]$)
(R3) NewNonce (n)
(R4) Send (S, $[i, \mathbf{N}, r, n]$)
(R5) Receive (S,$[\{\{i, \mathbf{N}, r, n, \mathbf{K}\}_{k_S^{-1}}\}_{k_r}]$)
(R6) Send ($i, [\{\{i, \mathbf{N}, r, n, \mathbf{K}\}_{k_S^{-1}}\}_{k_i}, \{n, \mathbf{N}\}_{\mathbf{K}}]$)
(R7) Receive ($i, [\{n\}_{\mathbf{K}}]$)
(R8) Accept ($\mathbf{K}$)
(R9) EndRespond (i)

Server (S) Protocol:

(S1) Receive ($r, [i, n, r, n']$)
(S2) NewSecret ($\{i, r\}, k$)
(S3) Send ($r, [\{\{i, n, r, n', k\}_{k_S^{-1}}\}_{k_r}]$)

Figure 2: Formal Specification Π_{KM}

The reader should consult [23, 24] for a more thorough presentation.

3.2 Protocol Specification

The formal specification of $\mathbf{\Pi}_{KM}$, denoted by Π_{KM}, is given in Figure 2. We explain the notation: SYS denotes the set of all principals in the system. In particular, Z, a distinguished principal who represents the intruder, and S, a principal representing the fixed server, belong to SYS. Lower case variables (e.g., i, r, p, n) range over *primitive* terms. In addition, i, r, p are assumed to range over names in SYS $-$ {S, Z}, while x, y over SYS. Bold face variables (e.g., $\mathbf{N}$) range over arbitrary non-null terms.

The translation of $\mathbf{\Pi}_{KM}$ to Π_{KM} is fairly straightforward. The key observation is that $\mathbf{\Pi}_{KM}$ actually consists of three distinct pieces, each of which is independently specified as a local protocol in Π_{KM}. The statements BeginInit, EndInit, etc. are added to facilitate the formalization of correctness. In particular, the Accept statements help differentiate session keys whose distribution have been completed from those that have not.

Among the initial conditions, (IC1) specifies that all principals know the names of every principal (including its own) and its public key. (IC2) specifies that the private key of a principal is known only to itself. (IC3) specifies that Z knows precisely the terms allowed under (IC1) and (IC2).

3.3 Correctness Specification

The mutual authentication goal is formalized using the two correspondence assertions below. (C1) specifies that whenever an initiator (i) finishes execution of its local protocol, it must be the case that the intended responder (r) has taken part in the exchange. (C2) specifies a similar property, but with respect to the responder.

$$(i, \text{EndInit}(r)) \hookrightarrow (r, \text{BeginRespond}(i)) \qquad \text{(C1)}$$

$$(r, \text{EndRespond}(i)) \hookrightarrow (i, \text{BeginInit}(r)) \qquad \text{(C2)}$$

In addition to the GSC, we specify the following secrecy assertion, (S), which says that the private key of each principal other than Z should not be learned by any other principal as a result of executing the protocol. Z is excepted because it is not bound by any protocol and is free to divulge its own private key.

$$\forall x, p : x \text{ has } k_p^{-1} \Leftrightarrow x = p \qquad \text{(S)}$$

The final correctness specification is $(\{(\text{C1}), (\text{C2})\}, \{(\text{S})\})$.

3.4 Proof Summary

First, we prove that Π_{KM} satisfies (S). Informally, we can see that (S) is satisfied because the private keys of non-server principals (except Z) are only used internally for decrypting incoming messages; they are never used in outgoing messages. For S, its private key is only used in encrypting messages and is never transmitted as a component in a message.

Lemma 1. Π_{KM} satisfies $\forall\ i, r, n, n', k$: $\neg(\text{Z has } \{i, n, r, n', k\}_{k_S^{-1}})$.

Lemma 1 says that Z can never learn a term of the form $\{i, n, r, n', k\}_{k_S^{-1}}$. Intuitively, this is because all such terms

are always sent out under the encryption of either k_i or k_r, and hence cannot be recovered by Z. Using Lemma 1, we can prove the following two lemmas.

Lemma 2.

$$(x, r, \mathsf{Comm}([\{\{i, n', r, n, k\}_{k_S^{-1}}\}_{k_r}])) \hookrightarrow (\mathsf{S}, y, \mathsf{Comm}([\{\{i, n', r, n, k\}_{k_S^{-1}}\}_{k_r}]))$$

Lemma 2 says that any message received by the responder at step (R5) must have originally been sent by S at step (S3).

Lemma 3.

$$(x, i, \mathsf{Comm}([\{\{i, n, r, n', k\}_{k_S^{-1}}\}_{k_i}, \{n, n'\}_k])) \hookrightarrow (r, y, \mathsf{Comm}([\{\{i, n, r, n', k\}_{k_S^{-1}}\}_{k_i}, \{n, n'\}_k]))$$

Lemma 3 is similar to Lemma 2. It says that any message received at step (I4) must have originally been sent by the responder at step (R6).

Lemma 4.

$$(x, r, \mathsf{Comm}([\{n\}_k])) \hookrightarrow (i, y, \mathsf{Comm}([\{n\}_k]))$$

Lemma 4 says that any message received at step (R7) must have originally been sent by the initiator at step (I5).

Using the above lemmas, we can prove that Π_{KM} satisfies GSC, (C1) and (C2). Thus we conclude that the protocol is correct.

Proposition. Π_{KM} is correct with respect to $(\{(C1), (C2)\}, \{(S)\})$.

4 Protocol Implementation

Our protocol implementation adopts the recently published GSS-API draft standard [11]. GSS-API is an implementation-independent interface through which security services are provided to callers. Specifically, it provides mutual authentication and per-message security services (e.g., confidentiality, integrity). Any application that makes use of the GSS-API interface can choose to use our protocol implementation as its underlying authentication mechanism. Our adoption of GSS-API facilitates the integration of our protocol implementation into any existing security architecture conforming to GSS-API.

Before we describe implementation details, it is important to emphasize that GSS-API is only an interface

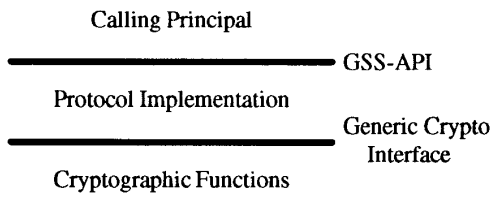

Figure 3: Structure of our Implementation

specification. In other words, it specifies only the *functions*, together with the semantics of their *parameters* and *return values*, that are available to a GSS-API user. In particular, many of the data structures (e.g., credentials, contexts) referenced are left unspecified at the interface level, and must be properly instantiated in an actual implementation.

We have also designed a generic interface for accessing all cryptographic functions needed to support our protocol implementation. This interface allows modular separation of protocol processing from cryptographic processing, thus facilitating easy substitution if a different cryptosystem is desired. We show the overall structure of our implementation in Figure 3.

Due to length limitation, we cannot cover every aspect of our implementation. Instead, we will focus our discussion on the mapping of our protocol steps onto GSS-API functions. In particular, we will discuss mainly *context level* calls, and mention *per-message* and *support* calls only to the extent that they are related. The ensuing discussion assumes basic familiarity with the design of GSS-API. The reader should consult [11] for clarification of any unfamiliar concepts.

Consider protocol Π_{KM} in Figure 1. The basic control flow of the initiator and responder is described below (Figure 4):

1. A principal must first acquire a set of *credentials* that allow verification of its identity by other principals. Credentials are used in the establishment of *security contexts* between authenticating principals, and are stored in *credential structures* internal to GSS-API. They are not directly accessible to a principal; instead they are referenced through opaque *credential handles*.

   Credentials can be acquired from various sources, e.g., smart cards, user files, or network. Typically, a credential contains the following items: name of the principal it certifies, name of the issuer, cryptographic keys, validity period, etc.

2. To establish a pair of security contexts (one each at the initiator and responder), the initiator and the responder must issue a pair of matching calls: GSS_Init_sec_context for the initiator and GSS_Accept_sec_context for the responder. Each

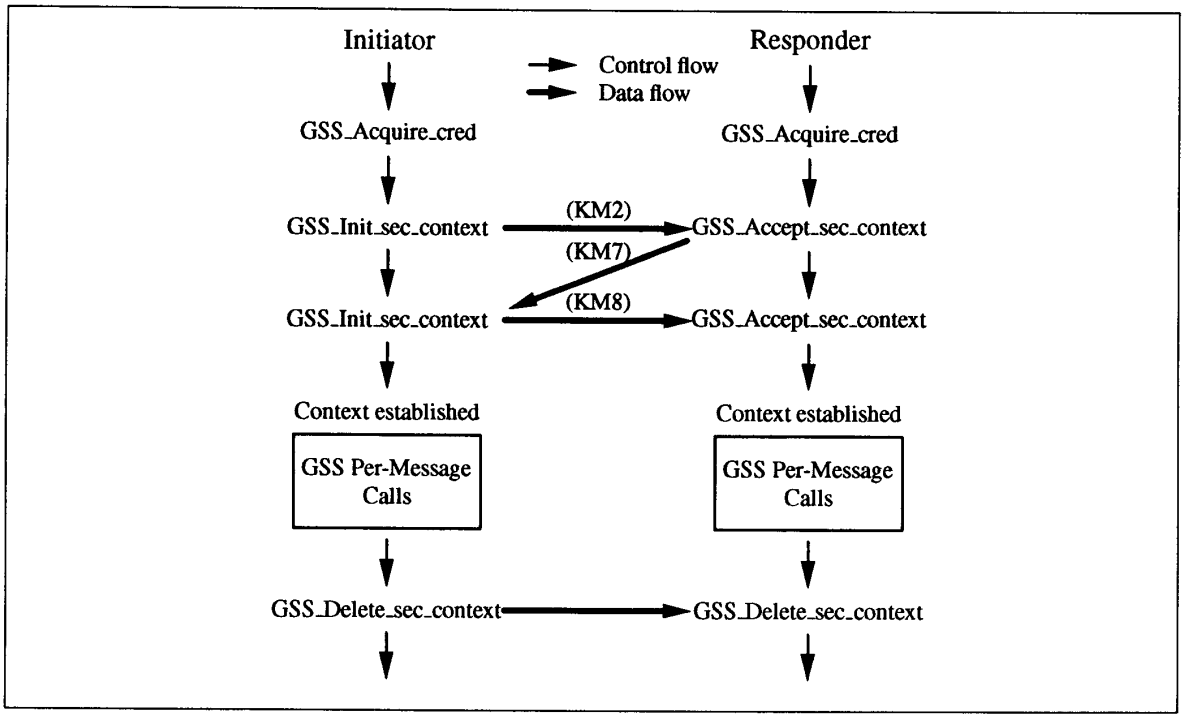

Figure 4: Control and Data Flow

function is called twice because mutual (as opposed to one-way) authentication is desired. These functions are responsible for generating the required protocol messages, called *tokens* in GSS-API, according to the protocol specification. The data flow arrows show the token flow between the two principals. The label of each data flow arrow corresponds to the protocol step label in Figure 1. Specifically, the token transferred over a particular data flow arrow encodes the protocol message in the corresponding protocol step. As we noted before, the data flows (KM2), (KM7) and (KM8) correspond precisely to steps of the three-way handshake protocol used in TCP connection establishment. These tokens are shown in greater detail in Figure 5 below.

We note that communications between the responder and the server are completely encapsulated within GSS_Accept_sec_context and hence do not appear as token flows in Figure 4. (A more detailed discussion of security context and tokens is given below.)

3. After a joint security context is established at both principals, per-message security services such as confidentiality, integrity and data origin authentication can be invoked. These services make use of information in a security context for their operations. For example, GSS_Seal uses the session key stored in the security context for its encryption operation.

   GSS-API provides the following per-message calls: GSS_Seal, GSS_Unseal, GSS_Sign and GSS_Verify. The functions of these calls should be self-explanatory.

4. At the completion of a communication session, a principal can destroy the associated security context by calling GSS_Delete_sec_context. This function returns a control token that can be passed to the other principal to effect a similar deletion of the corresponding security context.

Considering the lifecycle of a communication session, the control flow in Figure 4 can be broken down as follows: the GSS_Init_sec_context and GSS_Accept_sec_context calls belong to the connection establishment phase, the per-message calls belong to the data transfer phase and the GSS_Delete_sec_context call belongs to the connection teardown phase. Indeed, as we will see below, Π_{KM} is mapped completely within the GSS_Init_sec_context and GSS_Accept_sec_context calls.

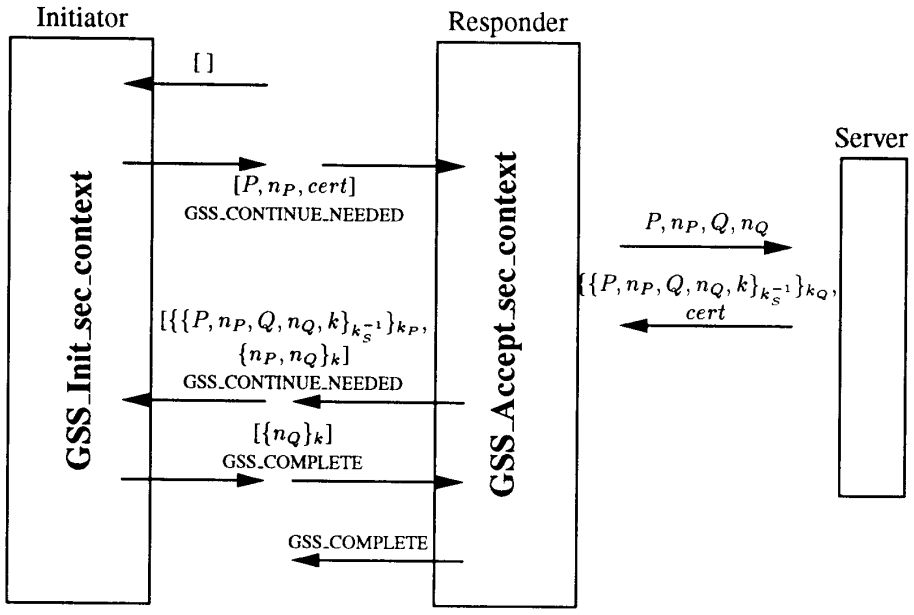

Figure 5: Mapping of Π_{KM} onto GSS-API

Consider Figure 5. A token is denoted by [...] (with [ ] denoting the NULL token). For the two boxes corresponding to GSS_Init_sec_context and GSS_Accept_sec_context, each arrow pointing inward into the box represents a *call* while each arrow pointing outward from the box represents a *return*. The label of each arrow represents either an input argument (for call arrows) or an output argument and return value (for return arrows). For example, consider the second arrow from the top of the GSS_Init_sec_context box, it represents a return from a call to GSS_Init_sec_context with token $[P, n_P, cert]$ as its output argument and GSS_CONTINUE_NEEDED as the return value.

One subtlety about GSS-API which may not be obvious is the fact that communications between the initiator and the responder are the responsibility of the (initiator and responder) principals who call GSS-API. The GSS-API functions only generate and process tokens, and do not handle the forwarding of tokens. In Figure 5, we illustrate this explicitly with disconnected call and return arrows. In our current implementation, this forwarding is via a socket channel. On the other hand, communications between the responder and the server are carried out within GSS_Accept_sec_context and are the responsibility of GSS-API. Our current implementation uses a UDP socket connection for this purpose. We will switch to the use *remote procedure call* (RPC) in the next version.

In our protocol design, we assume that the public key of each principal is known to every other principal. This assumption can be realized with several approaches: (1) Each principal can maintain a local file of all public keys it would ever use. (2) A separate public key retrieval protocol can be implemented, which retrieves a specified public key certificate from a *certificate depository* on demand. The certificate depository needs not be trusted and can be widely replicated. A *certificate cache* can also be maintained locally to improve efficiency. (3) Certificates can be piggybacked onto protocol messages. Either a *push* or a *pull* model can be followed. In a push model, the initiator piggybacks its public key certificate onto its first message to the responder (step (KM2)). In a pull model, the server piggybacks the public key certificate of the initiator onto its reply to the responder (step (KM6)). We illustrate both scenarios in Figure 5 (*cert* denotes a public key certificate).

Approach (1) is not satisfactory because of the difficulty in maintaining consistency and the lack of *a priori* knowledge of all the principals one wants to communicate. Our implementation is based on approach (3); but it falls back to approach (2) when a certificate is not locally available (e.g., following expiration).

We encode all protocol messages in XDR format [19]. This allows our implementation to be portable in a heterogeneous environment. We pick XDR over other representation schemes (e.g., ASN.1 encoding [2]) mainly for its simplicity and availability.

We choose respectively RSA [17] and DES [14] as the underlying asymmetric and symmetric cryptosystems. Both are *de facto* standards for their respective purposes. We use MD5 [16] as our message digest algorithm. In particular, it is used in the signing of all certificates. Lastly, we note that the inner encryption by the private key of S in steps (KM6) and (KM7) is actually implemented as a sign operation; this is sufficient because secrecy is already provided by the outer encryption under the respective intended receiver's public key.

5 Discussion

A systematic way of designing authentication protocols is to follow a stepwise refinement process (starting with small and intuitively correct protocols), while observing known design principles [1]. From our experience and that of others, such a process can help avoid many common protocol errors, and often provide protocols that are easier to verify. Much research is needed to identify useful design principles and refinement heuristics.

Given the current state of authentication protocol research, it does not appear that such a stepwise refinement process can be completely formalized. Certain refinement steps, however, appear to be easier to formalize than others. For example, in deriving Π_K from Π_3 in Section 2.1, it is apparent that the two protocols are essentially the same except that in one, certain messages are forwarded while in the other, they are sent directly. A notion of protocol *equivalence* which captures such variations can be proposed. Equivalent protocols satisfy similar properties and verification can proceed with the one that is easier to verify.

Concerning verification, one must bear in mind that the properties established hold only for the formal protocol, and may or may not hold for an actual protocol implementation (unless verification is carried out at the code level). Thus, it is desirable to use a verification methodology that minimizes the semantic gap between what is verified and what is implemented. In other words, the mapping between the formal protocol and its actual implementation should be relatively simple and intuitively justifiable. Woo and Lam's methodology is based on the use of transition-based semantics for both specification and verification, and allows a simple mapping from the formal protocol to the actual implementation. On the other hand, most logical approaches (e.g., [4, 6]) assume a very high level of abstraction, making them prone to misuse and misinterpretation. For example, BAN logic [4] application requires a protocol to be first *idealized*. The idealization process is at best informal, and at worst error prone. Indeed, there have been several cases where invalid conclusions were drawn from its application, mainly resulting from invalid idealizations [12, 15].

The verification in Section 3 is carried out on Π_{KM}, rather than on its components Π_K and Π_M. That is, the modularity of Π_{KM} is not exploited in its verification. It is worthwhile to investigate how the verification task of a "composed" protocol can be broken down so that the verification of its individual components can be reused.

Our current prototype implementation operates above the transport layer. Indeed, it makes use of services from the transport protocols (i.e., TCP and UDP). This restricts the users of our implementation to be application layer programs, e.g., TELNET, FTP. We plan to extend our implementation for use at the transport layer. The major consideration in such an extension would be performance. In particular, efficient credentials/contexts/cache management and connection demultiplexing are crucial. We will report our findings in a future paper.

The choice of XDR eliminates many of the portability concerns and simplifies some of our coding. However, XDR does have various quirks that hamper flexibility. In particular, the range of data types offered is fairly restricted. Both Kerberos and SPX adopt ASN.1. It would be interesting to compare and contrast the way XDR is used in our implementation with their use of ASN.1.

For cryptographic operations, we have been using some relatively inefficient software implementations of RSA and DES. Thus, we are not able to draw much conclusion about performance. In any case, our focus in this paper is on the connection establishment phase, performance is not as important a concern as in the data transfer phase (i.e., per-message calls). We hope to switch over to a highly optimized cryptographic package, and perform a more detailed analysis of performance. The design of a cryptographic interface for protocol support turned out to be a difficult task. For example, a design that optimizes related invocations of cryptographic functions could significantly improve performance. More research is needed.

6 Conclusion

We have provided a complete account of the development cycle of a realistic authentication protocol. Specifically, we discuss its design, specification and verification, and implementation. Hopefully, this global perspective on authentication protocol development would be a valuable reference to protocol designers, verifiers, and implementors.

The protocol was derived through a systematic stepwise refinement process. Although the process is informal, it is useful in light of the highly error-prone nature of authentication protocol design [1].

The protocol presented is part of an overall security architecture we have proposed. An implementation of the

basic protocol have been completed. An application programming interface for the protocol along with more details of the implementation are reported in [25]. We are currently integrating the protocol implementation with the rest of the security architecture.

Due to length limitation, we have omitted many practical details. We hope to include them in a future paper which describes the entire architecture. Many of the implementation details are as important from a security viewpoint as the overall design.

Research on authentication protocols is still a relatively young area. We believe that more experience of the sort reported in this paper is useful in bridging the gap between theory and practice and in improving our understanding of authentication protocols.

For future work, we are looking at systematic ways of extending intradomain authentication protocols such as the one presented in this paper to interdomain authentication needs. Again, a modular extension is preferred.

References

[1] M. Abadi and R. Needham. Good engineering practice for security in cryptographic protocols. Manuscript, November 1993.

[2] CCITT Recommendation X.208 Specification of Abstract Syntax Notation one (ASN.1), 1988. See also ISO/IEC 8824, 1989.

[3] R. Bird, I. Gopal, A. Herzberg, P.A. Janson, S. Kutten, R. Molva, and M. Yung. Systematic design of a family of attack-resistant authentication protocols. *IEEE Journal on Selected Areas in Communications*, 11(5):679–693, June 1993.

[4] M. Burrows, M. Abadi, and R.M. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[5] M.H. Cheheyl, M. Gasser, G.A. Huff, and J.K. Millen. Verifying security. *ACM Computing Surveys*, 13(3):279–339, September 1981.

[6] L. Gong, R.M. Needham, and R. Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings of 11th IEEE Symposium on Research in Security and Privacy*, pages 234–248, Oakland, California, May 7–9 1990.

[7] Information Sciences Institute, Marina Del Rey, California. *DARPA Internet Program Protocol Specification, Transmission Control Protocol*, September 1981. RFC 793.

[8] R.A. Kemmerer. *Formal Verification of an Operating System Security Kernel*. UMI Research Press, 1982.

[9] R.A. Kemmerer. Analyzing encryption protocols using formal techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448–457, May 1989.

[10] B. Lampson, M. Abadi, M. Burrows, and T. Wobber. Authentication in distributed systems: Theory and practice. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 165–182, Asilomar Conference Center, Pacific Grove, California, October 13–16 1991.

[11] J. Linn. *Generic Security Service Application Program Interface*, September 1993. RFC 1508.

[12] W. Mao and C. Boyd. Towards formal analysis of security protocols. In *Proceedings of The Computer Security Foundations Workshop VI*, pages 147–158, Franconia, New Hampshire, 1993.

[13] J.H. Moore. Protocol failures in cryptosystems. *Proceedings of IEEE*, 76(5):594–602, May 1988.

[14] National Bureau of Standards, Washington, D.C. *Data Encryption Standarad FIPS Pub 46*, January 15 1977.

[15] D. Nessett. A critique of the Burrows, Abadi and Needham logic. *ACM Operating Systems Review*, 24(2):35–38, April 1990.

[16] R. Rivest. *The MD5 Message-Digest Algorithm*, April 1992. RFC 1321.

[17] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978.

[18] J.G. Steiner, C. Neuman, and J.I. Schiller. *Kerberos*: An authentication service for open network systems. In *Proceedings of USENIX Winter Conference*, pages 191–202, Dallas, TX, February 1988.

[19] Sun Microsystems, Inc. *XDR: External Data Representation Standard*, June 1987. RFC 1057.

[20] J.J. Tardo and K. Alagappan. SPX: Global authentication using public key certificates. In *Proceedings of 12th IEEE Symposium on Research in Security and Privacy*, pages 232–244, Oakland, California, May 20–22 1991.

[21] T.Y.C. Woo and S.S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, January 1992. See also [22].

[22] T.Y.C. Woo and S.S. Lam. "Authentication" revisited. *Computer*, 25(3):10, March 1992.

[23] T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *Proceedings of 14th IEEE Symposium on Research in Security and Privacy*, pages 178–194, Oakland, California, May 24–26 1993.

[24] T.Y.C. Woo and S.S. Lam. Verifying authentication protocols: Methodology and example. In *Proceedings of International Conference on Network Protocols*, pages 36–45, San Francisco, California, October 19–22 1993.

[25] T.Y.C. Woo, B. Raghuram, S. Su, and S.S. Lam. SNP: An interface for secure network programming. In *Proceedings of USENIX Summer Technical Conference*, pages 45–58, Boston, Massachusetts, June 6–10 1994.