Secret Agents – A Security Architecture for KQML *

Chelliah Thirunavukkarasu
Enterprise Integration Technologies
Menlo Park
California CA 94025†

Tim Finin and James Mayfield
Computer Science and Electrical Engineering
University of Maryland Baltimore County
Baltimore MD 21228 USA

Abstract

KQML is a message protocol and format for software agents to communicate with each other. In this paper we discuss the security features that a KQML user would expect and an architecture to satisfy those expectations. The proposed architecture is based on cryptographic techniques and would allow agents to verify the identity of other agents, detect message integrity violations, protect confidential data, ensure non-repudiation of message origin and take counter measures against cipher attacks.

1 Introduction

Agents, in their different manifestations as filter agents, personal agents, softbots, knowbots etc, have become an important topic and is one of the primary research areas in the academia and the industry. These agents, to successfully interoperate with each other and share their knowledge, need a common interface standard.

KQML, Knowledge Query and Manipulation Language [1] is such a message format and protocol, which enables autonomous and asynchronous agents to share their knowledge and or work towards cooperative problem solving.

With the popularity of internet and the possibilities offered by the agent technology we can expect an explosion of agents in the internet. For KQML to be an effective agent communication protocol in such an environment, it should provide some means for agents to communicate in a secure manner to protect the privacy and integrity of data and to provide for the authentication of other agents.

In this paper we discuss a security architecture which would enhance KQML and allow KQML speaking agents to authenticate senders, verify message integrity and have a private conversation.

* This work was supported in part by the Air Force Office of Scientific Research under contract F49620–92–J–0174, and the Advanced Research Projects Agency monitored under USAF contracts F30602–93–C–0177 and F30602–93–C–0028 by Rome Laboratory.

† This work was done while the author was at the Univerisyt of Maryland Baltimore County

Submitted to the *CIKM'95 Intelligent Information Agents Workshop*, Baltimore, December 1995.

2 Functional Requirements

We arrived upon the following requirements for a KQML security model based on the analysis of the security models for Privacy Enhanced Mail [4], Corba [3] and DCE [5]. Interested readers are referred to [2], for a thorough treatment of security threats and mechanisms to counter them.

- *Independence of KQML and application semantics.* The security architecture should not depend on the semantics of KQML performative (i.e., an *ask-all* from an agent will entail a *tell* or *sorry* from the receiver. The security model should not rely upon this kind of interaction semantics). The security model should be general and flexible enough to support different models of agent interaction (e.g contract net, electronic commerce).
- *Authentication of principals.* Agents should be capable of proving their identities (they are who they actually claim to be) to other agents and verifying the identity of other agents.
- *Preservation of message integrity.* Agents should be able to detect intensional or accidental corruption of messages.
- *Protection of privacy.* The security architecture should provide facilities for agents to exchange confidential data.
- *Detection of Message duplication or replay.* A rogue agent may record a legitimate conversation and later play it back to disguise its identity. Agents should be able to detect and prevent such playback security attacks.
- *Non-repudiation of messages.* Non-repudiation of message is necessary to enforce accountability. An agent should be accountable for the messages that they have sent or received, i.e, they should not be able to deny having sent or received a message.
- *Independence of transport layer.* The security architecture should not depend on the features offered by the transport layer. This is critical to facilitate agents to communicate across heterogeneous transport mechanisms and to extend the security model to accommodate embedded KQML messages.

- *Non dependence on a global clock or clock synchronization.* The security architecture should not be clock dependent, as global synchronization of time is difficult to achieve and would lead to further security issues of its own. Further such a security model may have inherent security weaknesses [?].

- *Prevention of message hijacking.* A rogue agent should not be able to extract the authentication information from an authenticated message and use it to masquerade as a legitimate agent.

- *Authentication by crypto-unaware agents.* An agent need not have cryptographic capabilities to authenticate the sender of a message.

- *Support for a wide variety of crypto systems.* Agents should be able to use different cryptographic algorithms. But for two agents to interact, they should have a common denominator. The security architecture should not depend on any specific cryptographic algorithm.

3 Architecture Overview

The proposed security architecture is based on data encryption techniques [9]. In tune with the asynchronous nature of KQML, the model expects a secure message to be self authenticating and does not support any challenge/response mechanism to authenticate a message after it has been delivered. The architecture supports two security models, basic and enhanced. The basic security model supports authentication of sender, message integrity and privacy of data. The enhanced security model additionally supports non-repudiation of origin (proof of sending) and protection from message replay attacks. The enhanced security model also supports frequent change of encryption keys to protect from cipher attacks.

3.1 Definitions

The following paragraphs define the cryptographic techniques used by this architecture and the new performative and the parameters that have been introduced to implement the architecture.

Data Encryption Keys. An agent that implements the proposed security architecture should have a master key, K_a, which it would use to communicate with other agents. This key can be based on a symmetric or asymmetric key cryptosystem. If a symmetric key mechanism is used, we suggest that the agent, in addition to the general master key, also use a specific master key, $K_{a1,a2}$ for each agent that it communicates with, for better privacy and stronger authentication. If more than two agents share a single master key, any of those agents can masquerade as the other or eavesdrop on all the conversations between the agents sharing the key. If a master key is shared by more than two agents, the strength of security is directly related to the degree of trust between the agents.

If an agent does not share a master key, $K_{a1,a2}$ with another agent, it can use its master key, K_a, or can use the services of a central authentication server to generate such a key. The agents may use different keys in either direction of message flow i.e $K_{a1,a2}$ is created by *a1* and would be used when *a1* is sending a message to *a2* and $K_{a2,a1}$ is created by *a2* and would be used when *a2* is sending a message to *a1*.

If more than two agents share a single master key, any of those agents can masquerade as the other or eavesdrop on all the conversations between the agents sharing the key. If keys are shared by more than two agents, the strength of the security provided is directly related to the degree of trust between these agents.

If an asymmetric key mechanism is used, a unique key for each pair of agents is not necessary, as the agent can use the public key of its peer agent to encrypt the message and prevent eavesdropping. It can also use its private key to sign the message and prove its identity to its peer.

We assume that the agents know the master key of the other agents. We also suggest a secure mechanism to do master key lookup.

In the enhanced model, the agents use an additional key, the session key, to ensure privacy, message integrity and proof of identity. The session key can be symmetric or asymmetric and can be generated with the help of the authentication server. The session keys are set up by using a handshake protocol explained later. This handshake protocol requires the use of a master key to ensure security.

The agents can use either the session or master key for exchanging messages and must inform the other agent of the key that was used for encryption to ensure proper decryption.

When agents exchange keys, they encrypt them using the current session or master key. Keys are never exchanged in clear text form.

We recommend using the enhanced security model (if possible, as the enhanced model cannot be used under all circumstances) with an expensive master key and a cheap session key which is changed frequently.

Message Id. The message ID is used in the enhanced security model to protect agents from attacks by message replay. When the two agents establish a session key, they also exchange a message ID which the sender would use in the next message. Every message from an agent would carry a message ID and a new message ID for the next message. Each message ID is used only once to prevent replay and they are encrypted using the session or master key for security.

Message Digest. Each secure message generated using this architecture has a message digest or signature associated with it. The digest is calculated using a secure hash function like MD2, MD5 or SHS [9]. This hash function computes a digital fingerprint of the message (i.e acts as a "checksum" for the message). The sender then encrypts this digest using the session or master key and attaches it to the message.

This encrypted message digest forms the core of the security architecture. The receiver of a message uses this digest to verify the identity of the sender and the integrity of the message. The digest also protects the message ID field from being hijacked and used in a different message.

3.2 New KQML Parameters

The following new KQML parameters have been added to implement the security architecture.

:auth-master-key <boolean>. If T, the *:auth-digest* and *:auth-mesg* (if present) are encrypted using the master key. Else the session key is used. An agent would use the master key for encryption, if it does not share a session key with the receiving agent or if it does not know the receiver in advance. Under these circumstances, it could use this parameter to help the receiver in choosing the proper decryption key.

:auth-digest (<digest-type> <encrypted-digest>). The *digest-type* specifies the hashing function used (MD4, MD5, etc.) to compute the message digest. The *encrypted-digest* is the message digest encrypted using the key specified by the *:auth-master-key* parameter. This parameter should be present to prevent message hijack, and to provide for sender authentication and integrity assurance.

:auth-mesg-id <string>. The value of this parameter is a pre-agreed random string. This parameter is required only in the enhanced security model to prevent message replay. After verifying the current message, to prevent a reuse of the same message ID, the receiver should reset its internal message ID field to the *:auth-new-mesg-id* or NIL.

:auth-new-mesg-id <encrypted-string>. The value of this enhanced model parameter is the message ID for the next message and is encrypted using the key specified by the *:auth-master-key* parameter. For effective prevention of message replay, this parameter should be present in each message.

:auth-new-session-key (<key-type encrypted-key>). The value of this parameter specifies the session key for subsequent messages. If the value is T and the *:auth-shared-key* parameter is NIL, the current session key is destroyed and the sender will use the master key for subsequent messages. If the value is NIL, the session key, is left undisturbed. If it is not T or NIL, it is the new session key encrypted using the key specified by the *:auth-master-key* parameter. This parameter can be used to change the session key from time to time to protect from cipher attacks. Since the session key can be changed frequently, a cheap (computation-wise) cipher can be used as the session key.

:auth-mesg <encrypted-KQML-message>. This parameter is used only in *auth-private* performative. The value of this performative is a confidential KQML message which has been encrypted using the key specified by the *:auth-master-key* performative.

:auth-key-list ((<a1>, <key-type> <encrypted-key>) ...). This parameter is used by an agent during master key registration with the authenticator. The value is a list of 3-tuple. The first element is the agent name, the second element is the key type and the third element is the encrypted master key. If the agent name is NIL, that key is shared with all the agents. If an agent uses asymmetric master key, the parameter contains only key agent name set to NIL.

New KQML Performatives

The following new KQML performatives have been added to implement the security architecture.

auth-link. The sender wishes to authenticate itself to the receiver and set up a session key and message ID.

auth-challenge. The sender challenges the identity of the receiver in response to an *auth-link*. The sender encrypts a random string using the master key Ks,r or Ks and sends it as *:content*.

auth-link-request. The sender asks the receiver to send an *auth-link* and start the authentication process.

auth-private. The sender is sending a confidential message to the receiver. The *:auth-mesg* parameter contains the encrypted message and the *:auth-master-key* parameter specifies the encryption key. The *:auth-digest* parameter should be present to verify the identity of the sender and the *:auth-mesg-id*, *:auth-new-mesg-id* and *:auth-new-session-key* parameters may be present if enhanced security model is used.

auth-challenge-help. A crypto-unaware agent is enlisting the help of a trusted friend to construct a challenge message. The *:from* parameter will specify the agent to which the challenge message has to be sent.

auth-mesg-help. A crypto-unaware agent is enlisting the help of its trusted friend to authenticate a message with *:auth-digest* parameter. The message will contain the *:from* parameter whose value is the agent from which this message was received and the *:content* parameter's value will be the received message.

auth-key-request. The sender is requesting the authenticator to provide the master key for the agent specified in the *:from* field. If a master key pair exists for the two agents, the authenticator returns it. The *:content* parameter specifies the requested key's type. This performative can also be used to generate a master or session key. A key is generated if *:to* is used instead of *:from* and it is an error to use both. If *:to* is used, the *:content* parameter is a 2-tuple. The first element is the key-type and the second element is a boolean flag which will be true, if a master key is requested. If a master key is requested, the generated key is added to the key list of the sender.

4 Basic security model

An implementation should support the following protocol to conform with the basic security model. This model supports authentication, integrity and privacy of data. If asymmetric keys are used for session and master keys, this model also supports non-repudiation of origin.

When R2D2 sends a secure message to C3PO, it would compute a message digest and encrypt it using the master key.

```
<performative>
  :sender R2D2
  :receiver C3PO
  :auth-master-key T
  :auth-digest (<digest-type><encrypted-digest>)
  ...
```

Or, if R2D2 needs to send a confidential message to C3PO, it can encrypt the message and embed it in an *auth-private* performative.

```
auth-private
  :sender R2D2
  :receiver C3PO
  :auth-master-key T
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-mesg <encrypted-KQML-message>
```

This model can be used when R2D2 does not know the recipient in advance e.g. broadcast and facilitation performative. Or if R2D2 and C3PO do not require prevention of message replay and can afford the cost of using the master key.

In the above message, the *:auth-digest* parameter can be used to verify the integrity of the message, authenticate the sender and ensure non-repudiation of origin (if the master key is asymmetric in nature). If the message has been corrupted, the message digest will not agree with the value of the *:auth-digest* parameter. Since the message digest is encrypted with the master key of the *:sender*, only the *:sender* or the agents with which the *:sender* shares the encryption key could have generated the message. If the master key is an asymmetric key, only the *:sender* could have generated the message, as only the *:sender* knows the private key that has been used for encryption. Note that we can only verify the identity of the generator (i.e. the message was encrypted by the *:sender* agent) of the message. This message can be a replay of a legitimate message previously sent by the generator.

5 Enhanced security model

This model in addition to the basic security, supports prevention of message replay, and stronger non-repudiation of message origin (if asymmetric keys are used). Even though non-repudiation can be achieved in the basic security model, we can only be sure that the message was generated by the sender, as a rogue agent can replay a message and we will not be able to detect it.

In the remainder of this section we will demonstrate how the new KQML performatives and parameters can be used to converse/communicate securely.

5.1 Self authentication

Agent R2D2 has cryptographic capabilities and would like to prove its identity to agent C3PO. The agents would follow the following handshake protocol to achieve it.

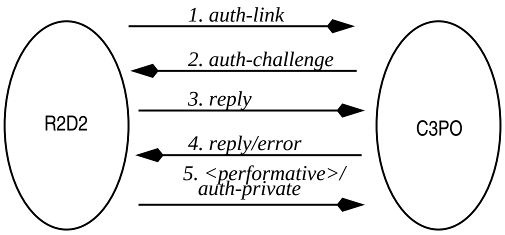

Figure 1: **Self authentication protocol**

1. R2D2 sends an *auth-link* performative to C3PO.

```
auth-link
  :sender R2D2
  :receiver C3PO
  :reply-with <expression>
```

2. If C3PO will not authenticate senders, it can respond with an *error*, otherwise it sends a *auth-challenge* with a random string encrypted using the master key. A random string is used to prevent message replay.

```
auth-challenge
  :sender C3PO
  :receiver R2D2
  :in-reply-to <expression>
  :reply-with <expression>
  :content <encrypted-random-string>
```

3. R2D2 responds with a *reply* performative with the *:auth-digest*, *:auth-new-mesg-id* and *:auth-new-session-key* (if present) encrypted in the master key. The value of *:content* and *:auth-mesg-id* is the decrypted random string. The session key parameter is optional.

```
reply
  :sender R2D2
  :receiver C3PO
  :in-reply-to <expression>
  :reply-with <expression>
  :auth-master-key T
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-mesg-id <mesg-id>
  :auth-new-mesg-id <encrypted-new-mesg-id>
  :auth-new-session-key (<key-type> <encrypted-key>)
  :content <random-string>
```

Now, C3PO can verify if the sender is R2D2 by inspecting the random string. Only R2D2 (or in the case of symmetric key, one of the other agents which shares the same key) could have decrypted the random string as it was encrypted using the master key. The message digest can be used for non-repudiation if asymmetric keys are used.

4. C3PO responds with a *reply* or an *error* depending on the success of authentication.

5. Now, R2D2 can send an authenticated message to C3PO by using the session key or master key to encrypt the message digest and a non replayable message by using *:auth-mesg-id* and *:auth-new-mesg-id* parameters.

```
<performative>
  :sender R2D2
  :receiver C3PO
  :auth-master-key T or NIL
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-mesg-id <mesg-id>
  :auth-new-mesg-id <encrypted-new-mesg-id>
  :auth-new-session-key (<key-type> <encrypted-key>)
  ...
```

Or if R2D2 needs to send a confidential message to C3PO, it can encrypt the message and embed it in an *auth-private* performative.

```
auth-private
  :sender R2D2
  :receiver C3PO
  :auth-master-key T or NIL
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-mesg-id <mesg-id>
  :auth-new-mesg-id <encrypted-new-mesg-id>
  :auth-new-session-key (<key-type> <encrypted-key>)
  :auth-mesg <encrypted-KQML-message>
```

5.2 Authentication by request

R2D2 may expect some of the incoming messages from C3PO to be authenticated and it can initiate the authentication process by following the handshake protocol given below:

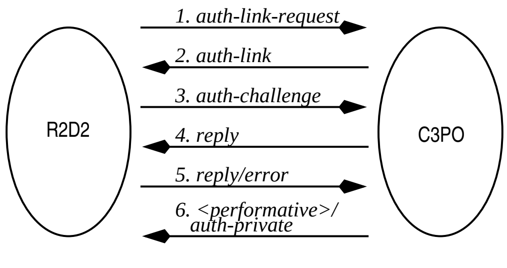

Figure 2: **Authentication by request protocol**

1. R2D2 can initiate the authentication process by sending an *auth-link-request* to C3PO.

```
auth-link-request
  :sender R2D2
  :receiver C3PO
  :reply-with <expression>
```

2. C3PO and R2D2 would then follow the steps outlined in *Self Authentication.*

5.3 Crypto un-aware agents

Agent Leia may not have crypto capabilities. But it trusts its friend R2D2 and R2D2 is prepared to authenticate messages on behalf of Leia. Since Leia does not have crypto capabilities, it will not accept *auth-private* performative. The agents would follow the handshake protocol given below to verify SkyWalker's identity.

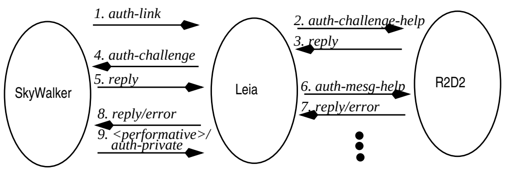

Figure 3: **Trusted friend protocol**

1. Agent SkyWalker sends Agent Leia an *auth-link* message to initiate the process of proving its identity to Leia.

```
auth-link
  :sender SkyWalker
  :receiver Leia
  :reply-with <expression>
```

2. When Leia receives an *auth-link* message from SkyWalker, Leia requests a challenge string from its trusted friend, R2D2.

```
auth-challenge-help\vspace{-0.30cm}
  :sender Leia
  :receiver R2D2
  :reply-with <expression>
  :from SkyWalker
```

3. R2D2 will generate a random string on behalf of Leia, encrypt it using the master key (shared by Leia and SkyWalker or Leia's master key, which R2D2 knows) and will forward it to Leia.

```
reply
  :sender R2D2
  :receiver Leia
  :in-reply-to <expression>
  :content (SkyWalker <encrypted-random-string>)
```

4. Leia will construct an *auth-challenge* performative and send it to SkyWalker. Subsequent performative from SkyWalker with an *:auth-digest* will be forwarded to R2D2.

```
auth-challenge
  :sender Leia
  :receiver SkyWalker
  :reply-with <expression>
  :in-reply-to <expression>
  :content <encrypted-random-string>
```

5. SkyWalker will respond with a secure *reply.*

```
reply
  :sender SkyWalker
  :receiver Leia
  :reply-with <expression>
  :in-reply-to <expression>
  :auth-master-key T
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-mesg-id <mesg-id>
  :auth-new-mesg-id <encrypted-new-mesg-id>
  :auth-new-session-key (<key-type> <encrypted-new-key>)
  :content random-string
```

6. Leia will wrap the response in an *auth-mesg-help* and send it to R2D2.

```
auth-mesg-help
  :sender Leia
  :receiver R2D2
  :reply-with <expression>
  :from SkyWalker
  :content message-from-SkyWalker
```

7. R2D2 will respond with a *reply* or an *error.*

8. Leia would forward the R2D2's reply to SkyWalker.

9. The handshake is now complete and SkyWalker can send secure performative to Leia, which Leia would verify with the help of R2D2.

6 Authenticator Agent

The authenticator acts as a repository of the agent's master keys. It can also generate session or master keys for the agents. The security architecture does not depend on the existence of an authenticator.

An agent and the authenticator share a master key which is known only to the agent and the authenticator. The master key may actually be a pair, one for the agent to send messages to the authenticator and the other for the authenticator to send messages to the agent.

The authenticator accepts only messages in the enhanced model, i.e., the messages should have an *:auth-mesg-id.* So, each agent should have established a secure link using *auth-link-request* and *auth-link* with the authenticator upon startup. It is the agent's responsibility to verify the identity of the authenticator and prove its identity to the authenticator.

6.1 Key lookup using the Authenticator

Agent Solo has received a message from Chewie and would like to know the master key used by Chewie. Solo uses the following protocol to get the master key from the authenticator.

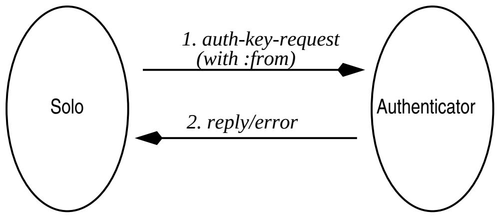

Figure 4: **Key request (lookup) protocol**

1. Agent Solo would send an *auth-key-request* to the authenticator to lookup the master key used by Chewie

to send out messages. The *:content* parameter contains the requested key-type.

```
auth-key-request
  :sender Solo
  :receiver Authenticator
  :reply-with <expression>
  :from Chewie
  :auth-master-key T or NIL
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-mesg-id <mesg-id>
  :auth-new-mesg-id <encrypted-new-mesg-id>
  :auth-new-session-key (<key-type> <encrypted-new-key>)
  :content <key-type>
```

2. If Chewie had previously registered a master key for communication with Solo, the authenticator will return that key in a *reply* performative. If there is no such key, the authenticator will reply with an *error*.

```
reply
  :sender Authenticator
  :receiver Solo
  :in-reply-to <expression>
  :auth-master-key T or NIL
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-mesg-id <mesg-id>
  :auth-new-mesg-id <encrypted-new-mesg-id>
  :auth-new-session-key (<key-type> <encrypted-new-key>)
  :content (Chewie <key-type> <encrypted-master-key>)
```

6.2 Key creation using the Authenticator

Agent Solo would like to send a secure message to Chewie and needs a session or master key for that purpose. It can send an *auth-key-request* to the authenticator to create such a key. If a master key has been requested, the authenticator would store the key in its database.

A master key creation would not be necessary if asymmetric keys are used as a single master key per agent is suffice to talk securely to all the agents. Further, non-repudiation of message origin is not possible if the authenticator knows the private key.

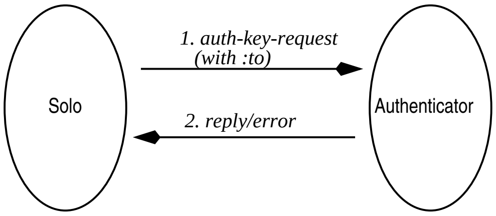

Figure 5: **Key request (creation) protocol**

1. Agent Solo would send an *auth-key-request* to generate a master or session key to send messages to Chewie. The *:content* parameter is a 2-tuple. The first element is the requested key's type and the second element is T if a master key is requested.

```
auth-key-request
  :sender Solo
  :receiver Authenticator
  :reply-with <expression>
  :to Chewie
  :auth-master-key T or NIL
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-mesg-id <mesg-id>
  :auth-new-mesg-id <encrypted-new-mesg-id>
  :auth-new-session-key (<key-type> <encrypted-new-key>)
  :content (<key-type> T-or-NIL)
```

2. Authenticator creates a key and sends it in a *reply* performative. If the requested key is a master key, the key is added to Solo's key list. If the authenticator is not able to create the key for whatever reason, it responds with an *error* performative.

```
reply
  :sender Authenticator
  :receiver Solo
  :in-reply-to <expression>
  :auth-master-key T or NIL
  :auth-digest (<digest-type> <encrypted-digest>)
  :auth-mesg-id <mesg-id>
  :auth-new-mesg-id <encrypted-new-mesg-id>
  :auth-new-session-key (<key-type> <encrypted-new-key>)
  :content (Chewie <key-type> <encrypted-master-or-session-k
```

6.3 Key registration with Authenticator using KQML

Agent Yoda would like to register its master keys with the authenticator.

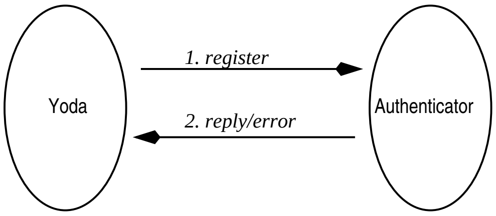

Figure 6: **Key register protocol**

1. Yoda would send a secure register with the keys in the *:auth-key-list* parameter. The keys are encrypted using the key specified by the *:auth-master-key* parameter. The agent can also use this performative to change the master key that it shares with the authenticator.

```
register
  :sender Yoda
  :receiver Authenticator
  :reply-with <expression>
  :auth-master-key T or NIL
  :auth-digest
    (<digest-type> <encrypted-digest>)
  :auth-mesg-id <mesg-id>
  :auth-new-mesg-id <encrypted-new-mesg-id>
  :auth-new-session-key
     (<key-type> <encrypted-new-key>)
  :auth-key-list
    ((<agent> <key-type> <encrypted-key>)...)
  :ontology tcp-address-ontology
  :content (tcp-host tcp-port)
```

2. If the key registration is successful, the authenticator responds with a *reply* else with an *error*.

6.4 Initial key registration with the authenticator

Agent Yoda is starting up for the first time and would like to register the master key that it shares with the authenticator. This can be achieved either using KQML or some other external mechanism.

If symmetric keys are used, KQML cannot be used to register the initial key as there is no master key to encrypt the key. If asymmetric keys are used, the initial master key is encrypted using the authenticator's public key. Even if asymmetric keys are used, there is a security problem. A rogue agent, agent DarthVader may know that agent Ben respects performative from agent Yoda. Agent DarthVader may also find out that Yoda has not registered with the authenticator and therefore the authenticator does not know the existence of such an agent. Now, DarthVader can register itself as Yoda. If this type of masquerading can be an issue, KQML should not be used for the initial registration.

The protocol would be same as the key register process. The *:auth-key-list* parameter will contain only one key pair and the agent name would be NIL as this is an asymmetric key and it is suffice to use a single asymmetric master key for all the agents.

If the authenticator does not have any entry for Yoda, it accepts the registration and adds it to its database and sends a *reply*.

7 Limitations of this model

The security model we are proposing for KQML has a number of limitation which vary in severity.

- An agent can send out authenticated messages if and only if it has crypto capabilities (A fair limitation).
- The security architecture introduces state information. Agent Emperor has to keep track of the next message ID and optionally the next session key that will be used by agent DarthVader. The agents can choose not to use this feature if they are not concerned with message replay attack and cipher attack.
- Messages delivery must be reliable and in order. (A fair limitation considering that KQML itself assumes that).
- The model does not support non-repudiation of receipt of messages. This would be difficult to implement due to the asynchronous nature of KQML and can be done only at the application level.
- There is no support for a mechanism to exchange credentials. Lets say that agent Emperor trusts agent DarthVader and would like to delegate DarthVader to act on its behalf. There is no way for DarthVader to take up Emperor's credentials.
- The model does not support replay detection if *:auth-mesg-id* and *:auth-new-mesg-id* are not used. These parameters cannot be used if the recipient is not known in advance.
- The architecture does not address traffic analysis by rogue agents. We feel that traffic analysis is best handled at the link/transport layers.
- The model should be enhanced to support the use of the Crypto APIs recommended by NSA (GSS, GCS and Cryptoki) [8], especially for the key-type and digest-type values.

8 Implementation

We have proposed a security model which has not yet been implemented. We have examined the modifications which would be required to integrate this model into some existing KQML API software. We will briefly discuss the approach to implementing this architecture in the KATS KQML API [10] software architecture.

in this architecture, each agent application is associated with its own separate router sub-agent. The routers used by all the agents are identical and handle all KQML messages going to and from the agent. The security enhancement can be easily added to this KQML implementation by modifying the router to be security aware, without involving any major change to the agent application.

The agent application only needs to specify the degree of security (any combination of provide for message authentication, protect from replay attack, send a confidential message and sign the message-non-repudiation of origin) of an outgoing message. The router would handshake with the receiving agent and secure the message to the extent possible (the receiving agent may not support asymmetric key cryptography, *auth-private* performative etc or the router may not know the receiving agent of the embedded message if it is sending out a broadcast or facilitation performative).

Similarly, when the router receives an authenticate request from another agent, it can handle the handshake itself, without involving the agent application. When the router receives a message from another agent, it would tag the message with a security level (confidential, authenticated, etc.). The agent application can decide to process or ignore the message based on the message's security level.

A similar approach can be followed to add security enhancement to most other KQML implementations. Most implementations would provide a library with at least a basic send and receive primitive to send and receive KQML messages. These primitives can be modified to add the authentication information to the outgoing messages or process the authentication information in the incoming messages. The implementations can use one of NSA recommended crypto APIs [8] for cryptographic capabilities. These APIs provide support for asymmetric and symmetric key cryptography, message digest, key generation etc. The use of a standard API would help agents using different KQML implementations to interact without any incompatibility problems.

9 Conclusion

The proposed security model addresses privacy, authentication and non-repudiation (if asymmetric key mechanism is used for the master and session keys) in agent communication. It does not fully address the issue of message replay, especially if the recipient of a performative is not known in advance. Ultimately, this security model depends on the strength of the crypto algorithm, message digest function and the random number generator used by the agent for its effectiveness.

10 Acknowledgments

This work has been the result of very fruitful collaborations with a number of colleagues with whom we have worked on KQML and other aspects of the Knowledge Sharing Effort. We wish to specifically thank and acknowledge Don McKay, Robin McEntire, Richard Fritzson, Charles Nicholas, Yannis Labrou, R. Scott Cost, and Anupama Potlori. Arulnambi

Kaliappan suggested the agent names and and Senthil Periaswamy of the University of South Carolina provided stimulating discussions on possible security threats and attacks.

References

[1] Draft specification of the KQML agent communication language, Tim Finin, Jay Weber et al, Jun 15 1993, http://www.cs.umbc.edu/kqml/kqmlspec/spec.html

[2] Security Mechanisms in High-Level Network Protocols, Victor L.Voydock, Stephen T. Kent, ACM Computing Surveys, Vol.15, No. 2, 135-171, Jun 83

[3] OSTF RFP3 Submission, Corba Security, OMG Document Number 95-3-3, Mar 8 1995, http://www.omg.org/docs/95-3-3.ps

[4] Privacy Enhancement for Internet Electronic Mail: Part I: Message Encryption and Authentication Procedures, J. Linn, Oct 02 1993, http://ds.internic.net/rfc/rfc1421.txt

[5] Security in a Distributed Computing Environment, OSF-O-WP11-1090-3, http://www.osf.org/comm/lit/OSF-O-WP11-1090-3.ps

[6] Project Athena Technical Plan, Section E.2.1, Kerberos Authentication and Authorization System, S.P.Miller, B.C.Neuman, J.I.Schiller and J.H.Saltzer, Oct 27 1988, ftp://athena-dist.mit.edu/pub/kerberos/doc/techplan.PS

[7] Limitations of the Kerberos Authentication System, S.M. Bellovin, M. Merritt, Proceedings of the Winter 1991 Usenix Conference, Jan 1991, ftp://research.att.com/dist/internet_security/kerblimit.usenix.ps

[8] Security Service API: Cryptographic API Recommendation, NSA Cross Organization, CAPI Team, Jun 12 1995, http://www.omg.org/docs/95-6-6.ps

[9] RSA Labs' frequently asked questions (FAQ), http://www.rsa.com/rsalabs/faq

[10] Software Design Document for KQML, Revision 3.0, Mar 1995, LORAL Corporation, Paoli PA, USA

[11] Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. KQML: an information and knowledge exchange protocol. In Kazuhiro Fuchi and Toshio Yokoi, editors, *Knowledge Building and Knowledge Sharing.* Ohmsha and IOS Press, 1994.

[12] Tim Finin, Don McKay, Rich Fritzson, and Robin McEntire. The KQML information and knowledge exchange protocol. In *Third International Conference on Information and Knowledge Management*, November 1994.

[13] Yannis Labrou and Tim Finin. A semantics approach for KQML – a general purpose communication language for software agents. In *Third International Conference on Information and Knowledge Management*, November 1994. Available as `http://www.cs.umbc.edu/kqml/-papers/kqml-semantics.ps`.

[14] R. Neches, R. Fikes, T. Finin, T. Gruber, R. Patil, T. Senator, and W. Swartout. Enabling technology for knowledge sharing. *AI Magazine*, 12(3):36–56, Fall 1991.

[15] Ramesh Patil, Richard Fikes, Peter Patel-Schneider, Donald McKay, Tim Finin, Thomas Gruber, and Robert Neches. The DARPA knowledge sharing effort: Progress report. In B. Nebel, C. Rich, and W. Swartout, editors, *Principles of Knowledge Representation and Reasoning: Proc. of the Third International Conference (KR'92)*, San Mateo, CA, November 1992. Morgan Kaufmann.

[16] Chelliah Thirunavukkarasu. A Security Architecture for KQML. Technical Report MS-EECS-95-nn, COmputer Science and Electrical Engineering Department, University of Maryland Baltimore County. August, 1995.