Andrew Weinrich
Computer Science 642
Final Project
Design Document
2007-05-14

1 Introduction

This document describes a Secure Publish and Subscribe protocol (SPS). It is adapted from work originally performed for the URGIS grid computing project [8], [7]. The project is located at `http://www.cs.northwestern.edu/~urgis/`.

This protocol allows for a publisher P to send data about events to one or more subscribers S_1, S_2, S_3, etc. This protocol has three goals:

1. Only one initial trusted communication between S and P should be necessary to use the subscription; all other communication should use encryption, signing, or other authentication to guarantee secrecy.
2. S should not be allowed to receive events that it is not subscribed to; S also should not be able to find out any information about what other parties are subscribing to events. Intermediary eavesdroppers should not be able to obtain any information about either events or subscribers.
3. Key and subscription management should be centralized so that only one server must be completely trusted.

The SPS protocol uses a central server to manage subscriptions, store public keys, and send out events. Subscriptions are not open, as with RSS feeds; all subscriptions must be approved by the publisher. This system does not allow users to filter subscriptions based on properties of the events.

2 Event Encryption System

The simplest way of distributing events would be establish a secure connection to each user. However, if events are large, and connections are publisher-initiated, this is likely to incur unacceptable overhead at the publication server. It would be more efficient to use some form of network-level many-to-one distribution, such as TCP multicasting, to send a single event message to multiple recipients. This protocol assumes that such a transport system is available to the publisher and will be used to distribute messages.

Using multicast requires that a single encrypted message be readable by multiple recipients, precluding the use of SSL. One alternative is to use a special key for each class of event, and give the event class key to the user when the subscription is first registered. This has the disadvantage of requiring the user to securely store one key for subscription, which may be a considerable burden if the user has subscriptions to many classes of events.

A better approach is to use a unique symmetric key for each message, and then use the public keys of the subscribers to encrypt this message key. Because the message key is small, the cost of using public-key encryption will be reasonable even if there are many subscribers.

We can put a header onto the message that contains the message key encrypted for each recipient. However, if we simply list out the keys, we would have to tag each encrypted key with the identity of the subscriber. This is a security flaw, as it allows subscribers to see who else received the message.

To fix this flaw, we can turn the list into a hash table, using the low 32 bits of the subscriber's public key as the hashing code. To handle hash collisions, each hash code will be associated with a "bucket" that contains the encrypted message keys for multiple subscribers.

In a normal hash table, the entries in a bucket consist of $(hashcode, value)$ pairs. In this case, $hashcode$ is a portion of the public key for the subscriber. If a PKI system is being used, this will allow eavesdroppers to determine which subscribers are receiving the message, and our hash table provides no additional security.

This problem can be solved by putting a unique, random message identifier in the header. Each bucket, instead of having $(hashcode, value)$ pairs, will simply have a list of encrypted values, $E_{K_{subscriber,public}}[ID_{message} \| K_{message}]$. After a subscriber has calculated which bucket his entry lies in, he will successively attempt to decrypt the values until he finds one that begins with the message ID. The remainder of the value will be the symmetric key for the

message contents.

For even distribution, a hash table should have a prime number of buckets. This will often cause the message key table to have more entries than there are subscribers. In this case, the publisher can simply fill the excess buckets with random data - to an eavesdropper, the "garbage" buckets will be indistinguishable from the ones that actually contain keys.

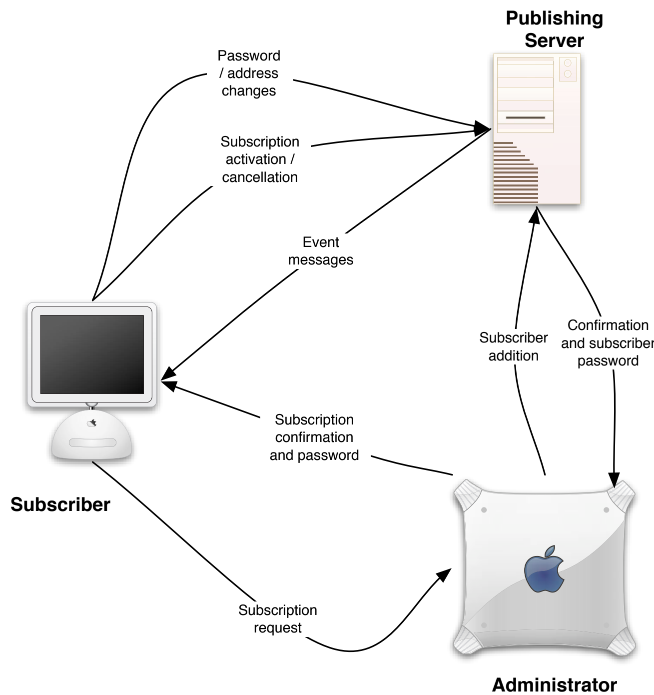
The format of these messages is described in the DTD listed in the appendix.

3 Protocol Entities

This following entities are used in this protocl:

- A publishing server P that serves events feeds for classes $E_1, E_2, \ldots E_n$. This server has a public key K_P that is used for signing messages.
- An administrator A who authorizes subscriptions to P's events. All communications from A to the subscribers are trusted.
- One or more subscribers S_i, who subscribe to various classes of events. Each subscriber S_i has a public key K_{S_i}. This key does not necessarily have to be registered with any kind of PKI system; it may be wholly private and used only by the subscriber.

4 Architecture Diagram



5 Message Flow and Usage Scenarios

5.1 Preparing for Publication

1. Bob, the administrator of a large VULTURE computing cluster, decides to establish a secure publish-and-subscribe system for notifying users of changes to the cluster. In this scenario, Bob himself serves the role of the administrator A.

2. Bob sets up a publishing server P, creates its RSA public/private key pair, and chooses an administrator password.
3. Bob adds several several classes of events, such as E_1="Added new nodes for operating system X", E_2="Upcoming downtime", etc.
4. Through secure channels, Bob lets potential subscribers know that the publishing service is available.

5.2 Subscribing to a Feed

5.2.1 Scenario

1. Alice, a scientist whose lab uses the cluster, wants to be notified when new Finux nodes are added. She sends a secure email to Bob with this request and her user ID.
2. Bob receives the request, and is assured that it comes from Alice. He logs in to the subscription server P, and adds the tuple $\langle Alice, Finux - Addition \rangle$. The server gives him a random subscription password N_{Alice}.
3. Bob send the password back to Alice by secure email.
4. Alice logs into the subscription server with her username and the password N_{Alice}. This communication is secured with SSL, though not necessarily with K_{Alice} and K_P.
5. The server checks Alice's id with the password N_{Alice} and authenticates her request. P stores K_{Alice} and the registration for event feed E_{Finux}. P then gives Alice the public key K_P that it will use to sign messages.
6. Alice tells the server the IP address of her subscriber system S, and optionally changes her password.
7. On her system S, Alice performs whatever setup is needed to receive messages from P (multicast reception, etc).

5.2.2 Message Flow

(1).	$\mathbf{S} \rightarrow \mathbf{A}$	$ID_S \| E_i E_j E_k \ldots$
(2).	$\mathbf{A} \rightarrow \mathbf{P}$	$ID_S \| E_i E_j E_k \ldots \| N_A$
(3).	$\mathbf{P} \rightarrow \mathbf{A}$	N_S
(4).	$\mathbf{A} \rightarrow \mathbf{S}$	N_S
(5).	$\mathbf{S} \rightarrow \mathbf{P}$	$ID_S \| N_S \| K_{S,public} \| E_i E_j E_k \ldots \| R_S$
(6).	$\mathbf{P} \rightarrow \mathbf{S}$	$K_{P,public}$

5.2.3 Message Rationale

ID_S	ID of subscriber S. This will be used for initial authentication to the publishing server P and activation of the subscription.
E_i	Identifier for event classes to which S is subscribing. Used by P to determine which messages to send to S.
N_A	The administrator password for the publishing server.
N_S	A password that S will use to prove its identity to P. This sent from P to S by using A as a trusted middleman.
$K_{S,public}$	Public key of S, used to ensure that event messages sent to S cannot be read by third parties. Does not need to be backed up with a certificate.
R_S	The internet address used by S for receiving messages.
$K_{P,public}$	The public key of the server P. Used by the publisher to sign message contents.

5.3 Publishing Events

5.3.1 Scenario

1. Bob decides that he need to send an update on channel E_{Finux}. He prepares the message contents D_E and sends it to the server P.

2. P looks up all of the subscribers who should receive E_{Finux}: $S_1, S_2, \ldots S_n$.
3. P generates a random message id ID_M and a symmetric message key K_M. It then creates a hash table that contains $E_{K_{S_i,public}}[ID_M \| K_M]$ for each subscriber. The hash table is padded with some extra entries to prevent eavesdroppers from guessing information about the receivers.
4. P computes the hash of the message ID and the event data, then signs this hash with its private key.
5. P uses the multicast system to distribute the message to all of the subscriber addresses.

5.3.2 Message Flow

(1). **A → P** $N_A \| D_{E_i}$
(2). **P → S** $ID_M \| Keytable(E_{K_{S_i,public}}[ID_M \| K_M], \ldots) \| E_{K_M}[ID_{E_i} \| D_{E_i}] \| E_{K_{P,private}}[H(ID_M \| D_{E_i})]$

5.3.3 Message Rationale

D_{E_i}	Message data for an event of class E_i.
ID_M	A unique identifier for this message.
$Keytable(\ldots)$	A hash table containing key entries for each subscriber.
K_M	A symmetric key used to encrypt the event data.
$E_{K_{S,public}}[ID_M \| K_M]$	An entry in the hash table. For each subscriber, the same value is encrypted with that subscriber's public key. No eavesdropper who is not a subscriber will be able to decrypt the message key.
ID_{E_i}	The identifier of this event class. Encrypted so that eavesdroppers cannot see what kinds of events are being distributed.
$E_{K_M}[ID_{E_i} \| D_{E_i}]$	The event class identifier and event data. Encrypted with a symmetric key for confidentiality.
$H(ID_M \| D_{E_i})$	A hash of the message ID and event contents, used for signing the message by the server.

5.4 Receiving Messages

5.4.1 Scenario

1. Alice's server S receives the event transmission from P.
2. S calculates its hash code from $K_{Alice,public}$ and looks up the appropriate hash bucket, which contains three entries.
3. The first entry in the bucket decrypts to garbage, but the second decrypts to $ID_M \| K_M$.
4. S uses K_M to decrypt D_E. It verifies that a hash of the message ID and event data is the same as that contained in the signature.

5.5 Other Tasks

If Alice ever needs to temporarily disable a subscription, add a new subscription, change the destination of events, or change her password or public key, she can establish a secured connection to the subscription server and send the appropriate commands.

6 Threat Assessment

The largest threat is that unauthorized parties will gain access to event data. It is also important to prevent attackers from finding out what kinds of events are offered. A subtler threat is that a third party will be able to gain information about events from watching the flow of messages from the publisher to subscribers. Knowing the complete list of subscribers for a type of event may allow some information about those events to be inferred.

This section describes how these threats are mitigated by the protocol.

6.1 Assumptions

The protocol makes several assumptions about communications and implementations:

- Local files stored on the publishing server are assumed to be secure. These includes DTDs for messages and the publishing database.
- The initial communication between a subscriber and the publishing administrator is assumed to be secure. This is required for the administrator to give the subscriber his initial account password.
- Subscribers are able to securely store private keys that they will use for
- The identity of the publishing server is trusted by the subscriber, either through assurance by the administrator or a signed identity certificate.
- The channel by which events are distributed from the publishing server to the users is *not* assumed to be secure. The ability of eavesdroppers to read the messages is addressed in the following sections.

6.2 Interception of Event Data

All event contents are encrypted with AES; the message key can only be obtained by using the public key of a subscriber to decrypt the appropriate entry in the message slot. Only subscribers will be able to decrypt the event data. Subscribers are assumed to be honest; i.e. that they will not intentionally distribute event data to other parties.

6.3 Interception of Other Information

Event feeds are not advertised by the subscription server; only be personally asking the publisher can a potential subscriber find out what events are available. It is assumed that this channel of communication is secure.

6.4 Inference of Subscriber Lists

The data in all slots of the key table is encrypted, so there is no way to tell which is a legitimate entry for a user and which is simply random data used by the publisher as padding. Since neither user IDs nor public keys are included, there is no way for a subscriber to determine the identity of other subscribers.

Because the key table for a message may contain more slots that there are subscribers, a single subscriber cannot know for certain how many other users subscribe to the same event. Additionally, an eavesdropper cannot assume that the size of the key list is an upper bound on the number of subscribers for that event; the publisher could easily create two different messages for the same event and distribute subscribers between them.

6.5 Snooping by Routers

It is possible that the intermediate routing nodes could construct the list of subscribers to an event by seeing which users receive copies of the same message. To protect against this kind of attack, the probabilistic multicast routing described in [5] can be used, so that no single routing node knows all of the recipients of a given message.

6.6 Impersonation of Administrator

The administrator is required to provide an admin password with every command that is sent to the publishing server. This password is extremely sensitive, and should be treated like this protocol's equivalent of the `root` password. Without knowing this password, or having physical/login access to the publishing server, there is no way for an attacked to impersonate the administrator and, for example, publish spurious messages.

6.7 Impersonation of Publisher

Depending on the method by which events are distributed, certain network attacks (such as DNS hijacking) may be able to make the subscriber think that a message has come from the server when it has actually been generated by a nefarious third party. This threat is covered by having the publishing server P sign the data of each event and include that signature in the message. Because the server's public key is known, an imposter will not be able to sign the message. This defense requires that either the public key be backed by a trusted certificate, or that the initial communication between the subscriber and publisher be trusted, so that the publisher's signing key can be exchanged securely.

6.8 Impersonation of Subscribers

Subscriber public keys are not necessarily backed up by certificates, which prevents the publishing server from automatically verifying the subscriber's identity. However, in order to register with the server and submit a key, the subscriber must log in with a password that the server itself generated. If the communications from the server to the administrator, and from the administrator to the subscriber, are secure, there is no way for an adversary to impersonate a subscriber.

6.9 Malformed Messages

All messages to the server are required to be in XML with a DTD reference. Messages are validated against the DTD before any processing occurs; if the message is not well-formed, an error is returned to the user. The DTDs are stored on the server and are assumed to be secure.

6.10 Message Signatures

Every message is identified with a 128-bit UUID. This allows subscribers to verify that new messages are in fact unique. The publishing server signs not just the event data for a message, but the concatenation of this unique message id and the event data. Even with identical event data, the same signature will not be used twice, preventing an eavesdropper from learning that identical events have been published.

7 Drawbacks and Limitations

There are several missing features and limitations to this Secure Publish-Subscribe protocol. None are insurmountable, but for the sake of time and simplicity this implementation does not attempt to solve them.

- The publishing server must be aware of all subscribers who will receive a message. Although there is no need for separate public keys for each message class, the publisher must store all users' keys and search its database every time it publishes a message. This causes the load on the server to scale linearly with the number of subscribers.

  Some systems, such as functional encryption [2], do not have this drawback; they allow for a single encryption of event data, with subscriber keys determining who is allowed to decrypt it. However, these protocols are more complicated than the simple version presented here.

- All communication with the publishing server is across HTTPS. The public keys of the publisher and subscriber are not exchanged using SSL, so they do not need to be backed up with a certificate chain. The SSL key used by the server should be signed to prove the server's authenticity, but a separate key may be used for signing the event messages. The subscriber is required to authenticate with his username and password, so there is no need to sign his public key.

  This separation of communications and signing keys is advantageous, as it is much easier for the subscriber to use a standard, unsigned, OS-provided HTTPS connection and manage its private subscription manually. For the administrator, it allows for the publishing server to be used hosted on a trusted third-party system, e.g. the CSL web server.

- Because the size of an RSA-encrypted block is larger than the plaintext block, the header for event messages will possibly be much larger than the actual event data, particularly if a large keytable that requires much padding is used, or if the actual event data is small.

8 Implementation Details

This section describes details of this particular implementation of the protocol, including drawbacks and design tradeoffs.

8.1 Architecture

Although the protocol is platform-agnostic, this implementation is targeted to Mac OS X. The code is written in a combination of several languages:

- Mac OS X has an implementation of the industry-standard CDSA security framework [4], a C API that provides cryptographic support for almost every major protocol. The C functions in this library are used to generate random numbers, create public keys, perform encryption and decryption, and sign messages.
- Objective-C is an object-oriented language that is used for most Mac OS X development. A custom-built Objective-C framework, CryptoLibrary, wraps the CDSA functions in simple classes called `AESKey` and `RSAKeyPair`, and provides easy serialization and deserialization. This library is largely derived from sample code provided by Apple [1]. The library also includes code for generating UUIDs and passwords, constructing and sending XML requests, and serializing databases.
- F-Script is a scripting language that allows interactive, interpreted use of Objective-C objects [3]. All end-user tools for the protocol are written in F-Script, which has a syntax very similar to Smalltalk.

8.2 Publishing Server

The publishing server is implemented as an F-Script CGI program. Apache is used as the HTTP server, with a self-signed SSL certificate. This setup considerably simplifies the design of the publishing "server" program, but does not scale well. This is not a flaw of the protocol, but merely of this demonstration implementation. There is also a short script that creates the publishing database; this script, which must be run once on the server, creates the publisher's signing key and the admin password, and sets up an empty subscriber list.

User passwords are stored in the server in plaintext. This is clearly a security flaw, but it aids in debugging and demonstration. In a production system, a hash of the password would be stored, in a similar manner as the Unix `/etc/shadow` password file.

8.3 Message Distribution

This implementation uses a world-readable directory as the publishing medium for messages. This is an alternative to TCP multicast, which would deliver messages directly to subscribers but is much more difficult to set up. Using the filesystem to publish events does not affect the security analysis, as it is already assumed that messages will be intercepted or possibly modified by eavesdroppers. Because each message has a unique identifier, subscribers can keep track of which messages they have already decoded, and only copy new messages from this directory.

8.4 User Interface

The user interface consists of two sets of F-Script scripts, one for the client and one for the administrator. The administrator does not require any support files, but a subscriber must run a setup script to create a public/private key pair, and create a small database file that will hold this key and the server's signing key. Otherwise, each script is a command-line program that sends a single command to the server.

8.4.1 Server Scripts

- `createpub.fs filename adminpassword` Creates an empty publishing database, and stores a brand-new server private key in it. The database itself is encrypted with a 128-bit AES key derived from the administrator password.
- `pub.fs` A CGI script that accepts incoming messages, as described in the "Document Type Definitions" section. This script requires that Apache set the environment variables `PUBLISHING_PASSWORD` and `PUBLISHING_DATABASE`.

8.4.2 Publisher Scripts

- `inspect.fs adminpassword` Returns a serialization of the current server state: the server's private key and a list of all subscribers. Subscriber entries include their activation flag, public key, password, list of allowed subscriptions, and list of active subscriptions.
- `publish.fs adminpassword subscription contents` Publishes an event to the given subscription. In this

  Returns a serialization of the current server state: the server's private key and a list of all subscribers. Subscriber entries include their activation flag, public key, password, list of allowed subscriptions, and list of active subscriptions.
- `adduser.fs server-url adminpassword username subscriptionlist` Adds a user to the database. The last parameter should be a comma-separated list of allowed subscriptions.
- `removeuser.fs server-url adminpassword username` Removes a user completely from the database.

The implementation also includes a GUI program, PublisherConsole, that combines the functionality of these scripts into

8.4.3 Subscriber Scripts

- `create-subscriber.fs username password` Creates a subscriber database with the filename `username.sub`. The script creates a brand new RSA key pair for the user, and stores it in the database along with the subscriber's username and account password. To protect this sensitive information, the database is encrypted with a 128-bit AES key derived from the password.
- `register.fs server-url username password` Sends a registration message to the server. This message includes the public key that was created with `create-subscriber.fs`. The server returns its public key, which the script stores in the `username.sub` file.
- `activate.fs server-url username password subscription` Sends a subscription activation message to the server.
- `deactivate.fs server-url username password subscription` Sends a subscription deactivation message to the server.
- `change-password.fs server-url username password new-password` Sends a password-change message to the server. This script also deletes the old `username.sub` file and

9 Document Type Definitions

9.1 Server Commands

Commands are sent to the publishing server by both the administrator and subscribers. The server uses the HTTPS protocol; the server and client public keys used for transport are not necessarily that same as those used for publishing messages.

Commands are sent as HTTP `POST` requests. The DTD for each command describes the XML that is sent as the request body. In the case of success, a code 200 is returnedThe following return codes are used:

1. `220 OK` - Returned when a request is completed successfully. Some commands will return data; the format and meaning of this data is described with each commands' DTD.
2. `400 Bad Request` - An error occurred. This will be returned if the command contained invalid XML, an invalid password was used, or another precondition was not met (e.g. attempting to activate an invalid subscription). The body contains a plaintext message describing the error.
3. `405 Method Not Allowed` - Returned is the user attempts to use a `GET` or other type of HTTP method besides `POST`.

9.2 Administrator Commands

9.2.1 Subscriber Addition

This command is sent by the administrator when adding a new subscriber. The response body contains, as plain text, the subscriber's randomly generated password.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Add a user to the database. On success, returns a randomly generated password. -->

<!ELEMENT removeuser (password, username, subscription+)>
  <!-- administrator password -->
  <!ELEMENT password (#PCDATA)>

  <!-- username -->
  <!ELEMENT username (#PCDATA)>

  <!-- Subscription that the user is permitted to receive. -->
  <!ELEMENT subscription (#PCDATA)>
```

9.2.2 Subscriber Removal

This command is sent by the administrator when removing an existing subscriber. Returns an error if the user does not exist.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Removes a user from the database. -->

<!ELEMENT removeuser (password, username)>
  <!-- administrator password -->
  <!ELEMENT password (#PCDATA)>

  <!ELEMENT username (#PCDATA)>
```

9.2.3 Event Publication

This command is sent by the administrator when publishing an event. Returns an error if no one is subscribed to the event.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Publish an event. -->

<!ELEMENT publish (password, subscription, event)>
  <!-- administrator password -->
  <!ELEMENT password (#PCDATA)>

  <!-- Subscription name. Server will determine who receives the publication. -->
  <!ELEMENT subscription (#PCDATA)>
```

```
<!-- Unformatted event data -->
<!ELEMENT event (#PCDATA)>
```

9.3 Subscriber Commands

9.3.1 Activation

Before a subscriber can receive publications, the account must be registered. This command includes a field for addressing information that would be used in TCP multicast or other distribution mechanisms.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for user registration. User must already have been added to the
     database by the administrator. -->

<!ELEMENT register (username, password, pubkey, address)>
  <!ELEMENT username (#PCDATA)>
  <!ELEMENT password (#PCDATA)>

  <!-- RSA public key. Will be used for encrypting message keys -->
  <!ELEMENT pubkey (#PCDATA)>

  <!-- Address used for sending publications. Required but ignored. -->
  <!ELEMENT address (#PCDATA)>
```

9.3.2 Subscription Activation

Subscriptions are not automatically active. Subscribers must individually activate every subscription they wish to receive. Returns an error if the requested subscription is not allowed.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for user subscription activation. The user must be permitted to the
     subscription and not yet have activated it. -->

<!ELEMENT activate (username, password, subscription)>
  <!ELEMENT username (#PCDATA)>
  <!ELEMENT password (#PCDATA)>

  <!-- Subscription name -->
  <!ELEMENT subscription (#PCDATA)>
```

9.3.3 Subscription Deactivation

If a subscriber wishes to temporarily disable a subscription, this command may be used. Returns an error if the subscription is not currently active. The "Subscription Activation" command can be used to re-activate the subscription.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for user subscription deactivation. The user must be permitted
     to the subscription and have activated it. -->

<!ELEMENT deactivate (username, password, subscription)>
  <!ELEMENT username (#PCDATA)>
  <!ELEMENT password (#PCDATA)>

  <!-- Subscription name -->
  <!ELEMENT subscription (#PCDATA)>
```

9.4 Event Message

This DTD describes the format of event messages.

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- DTD for a published event -->

<!ELEMENT event (header, eventdata)>

  <!-- Header that contains a unique messageid, list of encryptions of the
       message key, and signature -->
  <!ELEMENT header (messageid, keylist, signature)>

    <!-- Unique, random ID for this message -->
    <!ELEMENT messageid (#PCDATA)>

    <!-- Simple hash table for key entries. The low 32 bits of each subscriber's
         public key are used as indices into the table -->
    <!ELEMENT keylist (keybucket*)>
      <!ELEMENT keybucket (keyentry*)>

        <!-- A keyentry contains the concatenation of the messageid and the message key,
             encrypted with a subscriber's public key -->
        <!ELEMENT keyentry (#PCDATA)>

    <!--Hash of the messageid+eventdata, signed with the publisher's private key -->
    <!ELEMENT signature (#PCDATA)>

  <!--Actual data for the event, encrypted with the symmetric message key -->
  <!ELEMENT eventdata (#PCDATA)>
```

References

[1] Apple Computer, CryptoSample framework, `http://developer.apple.com/samplecode/CryptoSample/index.html`, 2003.

[2] Vipul Goyal, Omkant Pandey, Amit Sahai, and Brent Waters, "Attribute-Based Encryption for Fine-Grained Access Control of Encrypted Data", *Proceedings of 13th ACM Conference on Computer and Communications Security*, 2006.

[3] Phillipe Mougin and Stephane Ducasse, "OOPAL: Integrating Array Programming in Object-Oriented Programming", *OOPSLA 2003 Conference Proceedings*, 2003.

[4] The Open Group, "The Common Data Security Architecture", `http://www.opengroup.org/security/l2-cdsa.htm`, 2001

[5] L. Opyrchal and A. Prakash, "Secure distribution of events in content-based publish subscribe systems", *Proceedings of the 10th USENIX Security Symposium*, 2001.

[6] Mudhakar Srivatsa and Ling Liu, "Secure Event Dissemination in Publish-Subscribe Networks", 2006.

[7] Andrew Weinrich, "Secure Multipoint Message Delivery Protocol", `http://pages.cs.wisc.edu/~weinrich/`, 2003.

[8] Andrew Weinrich, "Content Delivery Network", `http://pages.cs.wisc.edu/~weinrich/`, 2003.