Security in ad Hoc Networking
Team D
Secure Group Communication - Design

Jeremy Balmos
(jcb7565@cs.rit.edu)

Shawn Chasse
(smc0857@cs.rit.edu)

Jeremy Dahlgren
(jad0883@cs.rit.edu)

Eric Ferguson
(etf2954@cs.rit.edu)

December 19, 2004

Contents

1 Introduction

This paper introduces a implementation of a protocol suggested by Adrian Perrig in his *Efficient Collaborative Key Management Protocols For Secure Autonomous Group Communication* paper.

1.1 Adrian Perrig Communication Paper Review

Adrian Perrig presented a paper on *Efficient Collaborative Key Management Protocols for Secure Autonomous Group Communication*. In this paper he established three protocols for secure group communication over any type of network media.

Perrig described five basic properties that any *group key agreement protocol* (GKAP) should have. Any GKAP should be contributory, have implicit key authentication, have key integrity, provide key confirmation, and provide key independence. These properties provide perfect backward and forward secrecy. For more information about these properties refer to the background section in his paper.

Before presenting the protocols, Perrig stated several assumptions that the protocols will function under. One assumption is that after a group member has been authenticated it will not be malicious in any way. Another assumption is that each group member trusts a Certificate Authority or Key Authentication Center being used. A final assumption is that any group member fails by halting.

The first protocol presented does have any type of user authentication. A binary tree is used to give structure to the group. Each group member is located at a leaf in the binary tree. A group key is established by using a key generation procedure based on Diffie-Hellman key agreement between nodes. Keys for intermediate nodes are established by randomly selecting a node from the left subtree and right subtree to execute the Diffie-Hellman key generation procedure. The key established at the root node is the group key that will be used for secure communication. Since this protocol uses Diffie-Hellman to generate a key, it is vulnerable to a man in the middle attack. Even though this protocol lacked group authentication, it is still usable as a base for the other protocols to build upon.

Next he presented a protocol, which incorporates user authentication. This authentication is generated through a trusted third party or a certificate agency. The authentication is an authenticated Diffie-Hellman scheme using digital signatures and a public key infrastructure. The author went on to note that there is a disadvantage to using this type of authentication. First, "... for each key agreement, the certificates

need to be exchanged, which consumes bandwidth because of the large size of certificates." Furthermore "... the signature of the PKI [public key infrastructure] needs to be verified, which is computationally expensive." Based on this disadvantage, the author decided to present a third and final protocol.

The third protocol provided in this paper is based on *Gunther's identity based key agreement protocol*. Gunther's authentication strategy uses a key authentication center to distribute private and public information for each group member which is used in member authentication prior to any key agreement procedure. The workings of Gunther's protocol can be found in Perrig's paper and do not need to be repeated here.

Our implementation of secure group communication is based on Perrig's third proposed protocol. The exact key agreement was changed because of mathematical reasons. We ended up using a Diffie-Hellman based key agreement procedure which follows Perrig's algorithm for generating the key up the binary tree. The inner workings of our protocol implementation are described in the proceeding paper. Please reference the attached copy of Adrian Perrig's paper if there are any further questions about his protocols.

2 Design Notes

- A unique member in the group is called the *Group Controller (GC)*. The GC is responsible for administration of group members as well as initializing/synchronizing the key agreement procedure.
- The `BinaryTree` class is an array based binary tree, see section 5.1.
- A group has both a secure channel, see section 7.2, for private group communication, as well as an insecure channel, see section 7.1, for trivial group communication and management.

3 Design Flaws

- The initialization of the `SecureChannel` should be done within an additional thread. This would free up the `InsecureChannel` thread that is waiting on new messages, so they are not lost.
- Our lack of knowledge of M2MP prevented us from writing protocols that effectively used its functionality.

4 Known Problems

The following scenarios have not been handled yet. In our ongoing attempt to provide a user with the functionality to communicate securely with more than one person, the following will be addressed. At present, the outcome of the following scenarios will produce unexpected results.

- A timeout occurs during the key agreement procedure
- Two users attempt to create a group/join a group at the same time.
- A member joins the group during the key agreement procedure.
- A member leaves the group during the key agreement procedure.
- More than $2^{32} - 2$ members join the group, since we keep track of members by their member index, stored by an integer, and we reserver 0 and -1 as special cases. The integer is just incremented every time a member joins the group, but leaving member's indexes are not remembered and reused.

5 Binary Tree

All members of the group are kept in the `BinaryTree`. Every position in the tree is kept with a `BinaryTreeNode`.

5.1 BinaryTree

The only resemblance this class has to a standard binary tree is that, at most, any node is allowed to have 2 children. The `BinaryTreeNode` objects are kept in an array. By putting the tree in an array, transmitting the structure of the existing tree, to newly joining members, was very easy. As an array based binary tree, Perrig's algorithm for key agreement was also much easier.

When a new member is inserted into the tree, the tree tries to insert the new node as close to the root as possible. Members can only be leaves, so every node that is not a leaf has to be an empty node with the member index of -1, symbolizing that it's an intermediary node. Also, anytime a new level is added, the entire level (breadth-wise) is filled with place holders. These place holders have a member index of 0 and contain no data.

The `BinaryTree` class also provides the functionality to store a member's key based

on level of choice. That way the `Group` can store the intermediate keys during the key agreement procedure

5.2 BinaryTreeNode

A `BinaryTreeNode` is nothing more than a wrapper that holds two values. A `java.math.BigInteger` that is used as the key for the current node and an `int` that is used as the member index. The member indexes 0 and -1 are reserved for internal use.

0 A placeholder. There is no data here, but `java.util.Vector` cannot contain *holes*, so in order to use it the way we wanted, placeholders must be used.

-1 An intermediate node. Because our members can only be in leaves, we need intermediary `BinaryTreeNode`s on every level above the leaves. This is where keys are placed.

6 Group

The CPU/brain of our Secure Group Communication package would be the `Group` class. All of the functionality that is provided by the Secure Group Communication package is accessed, by the user, through the `Group` class. All of the other classes in this package are hidden from the user.

The `Group` will work directly with the `InsecureChannel`, see section 7.1, to carry out the setup of the group. When a user requests to join a group, the `Group` will send out a packet looking for the Group Controller. If the response times out the user will create the group. If the user gets back a response from the GC, then the user knows that the group exists and that he is not the GC. The GC will also send the new user a list of the current members and their respective positions in the tree, see section 5.

6.1 Group Public Signature

The signature of `Group` is the signature of the Secure Group Communication, since all functionality is accessed through the `Group` class.

`Group( String gn, MessageReceived ref )`	Join/Create the group *groupName*. Pass yourself as a reference for callbacks upon received messages
`leave()`	Leave the group
`sendInsecureMessage( String msg )`	Send *msg* over the `InsecureChannel`
`sendSecureMessage( String msg )`	Send *msg* over the `SecureChannel`

6.2 Heartbeat

Because of an ad hoc environment's dynamic nature, you must check to see that every member in the group is still *alive*, that is to say, they are still present. We achieve the assurance, of all members being *alive*, by having every member send a `Heartbeat` to every other member, every `HB_delay` seconds (section 6.3). Every member then checks on the status of the other members every `HB_check_delay` seconds (section 6.3). If any member has been idle (i.e. we have not received a `Heartbeat` from them) in `HB_TO` seconds (section 6.3), then we will notify the GC and remove that member from their `BinaryTree`.

6.3 Group Performance Tuning

Another constructor is available to the user to tailor the timeouts of the `Group` class.

6.3.1 Constructor

```
Group( String           groupName,
       MessageReceived  applicationRef,
       int              join_TO,
       int              HB_send_delay,
       int              HB_check_delay,
       int              HB_TO,
       int              key_TO )
```

`join_TO`	Number of milliseconds to wait before join call timeout and the group is created
`HB_send_delay`	Number of milliseconds to wait in between heartbeat sends
`HB_check_delay`	Number of milliseconds to wait in between checking the status of other members
`HB_TO`	Number of milliseconds before assuming a group memeber is dead and expelling them
`key_TO`	Time to wait before assuming a group member is dead during key agreement

6.3.2 Accessors and Mutators

All of these values, can be changed on the fly using the following accessors and mutators.

`void setJoinTimeout( int t )` `int getJoinTimeout()`	Change or acquire current `join_TO`
`void setHeartBeatDelay( int t )` `int getHeartBeatDelay()`	Change or acquire current `HB_send_delay`
`void setHeartBeatCheckDelay( int t )` `int getHeartBeatCheckDelay()`	Change or acquire current `HB_check_delay`
`void setHeartBeatTimeout( int t )` `int getHeartBeatTimeout()`	Change or acquire current `HB_TO`
`void getKeyProcTimeout( int t )` `int setKeyProcTimeout()`	Change or acquire current `key_TO`

7 Communication Channels

7.1 InsecureChannel

The `InsecureChannel` is used for all group management. When a user joins a group, since key agreement has not yet taken place, the communication must be done insecurely. After a user has entered a group, then key agreement takes place, and is also done over the `InsecureChannel`. After key agreement has been completed, all subsequent group communication is done over the `SecureChannel`.

7.1.1 TimeoutException

The `TimeoutException` is thrown by the blocking `receiveMessage( long timeout, int address )` call in `InsecureChannel` after *timeout* milliseconds.

7.2 SecureChannel

All secure group communication is done over the `SecureChannel`. Once a group key has been established, through key agreement procedure carried out over the `InsecureChannel`, the `Group` class initializes the `SecureChannel`. The `SecureChannel` then uses the `MultiKeyGenerator` to encrypt and decrypt messages as needed.

7.2.1 MultiKeyGenerator

The `MultiKeyGenerator` is used by the `SecureChannel` to encrypt and decrypt messages. The `MutliKeyGenerator` uses password based encryption with MD5 and DES, see **Java™Cryptography Ex (JCE) Reference Guide** for more information. The `MultiKeyGenerator` removes the knowledge about how the data is encrypted from the `SecureChannel`. Abstracting the data encryption and decryption from the `SecureChannel` is useful because it takes out complexity from the channel, and also decouples the encryption from itself.

8 Future Work

- Change implementation of key agreement from current state to use Gunther's key agreement.
- Rework protocol structure so that it is optimized for speed.
- Create the SecureChannel in a separate thread so that it will not block the group object from receiving insecure messages during key generation.
- Address the problem of timeouts during the key agreement procedure.
- Manage the group member indexes so that numbers may be reused if the necessity arises.
- If a member leaves during key agreement, cancel key agreement and process leave.
- If a member joins during key agreement, cancel key agreement and process join.
- Test our protocol in a real ad hoc environment.