Security Services for Group-Oriented Applications*

Bruno Dutertre

System Design Laboratory
SRI International
Menlo Park, CA 94025

March 21, 2000

*This work was partially funded by the UK Defence Evaluation and Research Agency.

Abstract

High-speed distributed networks have the ability to support groupware applications. In such applications, distant users cooperate and communicate to accomplish a common task. This paper discusses existing protocols, systems, and architectures proposed for securing group communication, and examines the intrusion-tolerance properties that these systems are intended to provide. In particular, we examine two lightweight frameworks supporting secure group applications, namely, Antigone [12] and Enclaves [5, 6].

Contents

1 Introduction

A group-oriented application enables users to share information and collaborate via a communication network such as the Internet. The group involved is usually dynamic. An application is typically started when a user initiates a session, and new users are allowed to join and later leave the session. Multicast is the main mode of communication in such applications: Messages originating from one of the group members are received by all the other users in the sessions. Point-to-point communication between two users can also be used, allowing users to share information independently of the rest of the group.

The security requirements of such applications can vary widely. In open applications, such as multicast of video or radio programs, a stream of data originates from a single information source in the network. Anybody can join the application and access the data. The data producer does not know and has no control over who has access to the information. The data distributed is not confidential and group control is minimal.

In many cases, groupware applications need more restricted access policies and have other security requirements. Access to an active session is limited to a predefined set of users or to users having appropriate credentials (e.g., they have paid for the service provided). In such contexts, user authentication, key distribution, data confidentiality, and data integrity are essential. Different systems and architectures can provide the necessary security infrastructure to support such applications. Examples include lightweight frameworks such as Enclaves [5] or Antigone [11, 12], or more complex systems such as Rampart [16] or the SecureRing Protocols [8].

The services provided by these infrastructures can be organized in two broad categories: communication and group management. The communication services ensure that users can communicate in a secure and reliable way. The group-management services perform user authentication, access control, and related functions such as key distribution.

1.1 Communication Services

Most groupware applications rely on multicast services to efficiently distribute messages originating from one group member to all the others. Ensuring the confidentiality and integrity of the communication is a basic requirement in many applications. For this purpose, standard techniques based on encryption, digital signatures, hash functions, or other mechanisms are used. The objective is to prevent outsiders from accessing or corrupting information in transit over the network. Insiders (i.e., the group members) are trusted and usually have access to all the information exchanged. Some protection against nontrustworthy insiders can be provided at the session-management level, for example, using procedures for excluding misbehaving members from the group.

Two common implementation approaches exist for such secure communication services: either security is added to an existing multicast protocol or secure multicast is implemented using standard point-to-point protocols. The first approach is usually simpler. In addition, one may inherit some of the properties of the underlying multicast protocols, such as reliability and, in some cases, atomic delivery of

messages[1]. However, multicast is not available everywhere and reliable multicast protocols tend to be complex and costly. Lightweight approaches such as Enclaves rely on a best-effort strategy and implement secure group communication using point-to-point protocols. No strong guarantees are provided on the order of delivery of messages to different members of the group.

Many of the existing protocols for reliable multicast (cf. [13] or [10] for surveys) were not designed to resist deliberate attacks but only unintentional network or node failures. Thus, it is not clear whether such protocols are a good basis for building secure applications. Systems such as Rampart or SecureRing use their own protocols, designed to be very reliable and resilient to certain classes of Byzantine failures [7,17].

1.2 Group and Session Management

Group- and session-management services form the core of any secure group-oriented application. These services are the primary protection against intrusions and are necessary to implement secure communication. As a minimum, such services include user authentication, access control, and key distribution.

In systems such as Enclaves [5] or Antigone [11], group management is centralized. A specific (and fixed) member of the group is the session leader and is responsible for authenticating and accepting new members, and for generating and distributing encryption keys. The session leader can also perform other functions such as expelling members or choosing the access control policy. For example, an access control policy may specify the objects that are globally shared by the members, and the access right of each member to these shared objects. The session leader usually initiates the application and also terminates the session.

The advantage of such an architecture is simplicity. The session leader provides protection against external attacks via the authentication services. It also provides a degree of protection against insider attacks via the access control mechanisms and the expelling mechanisms. Membership information can be easily collected and distributed to the members. On the other hand, such an architecture is not scalable and the session leader is clearly a single point of failure. In addition, regular users have little protection against a session leader that abuses its privileges. The leader can also have access to long-term user keys that a single intrusion at the leader node can compromise. In summary, a centralized architecture can provide good protection against outsider attacks and some protection against misbehaving insiders, but there is little protection against a failure or compromise of the session leader.

To provide better intrusion tolerance, systems such as Rampart [16] or the SecureRing [8] do not use a centralized architecture. Instead, they rely on fault-tolerant protocols to distribute all decisions about group management. Centerpieces of such systems are intrusion-tolerant distributed agreement protocols. The algorithms proposed by Reiter [14, 15] and by Kihlstrom et al. [7] ensure agreement between the nonfaulty members of the group, provided less than one third of the group members are compromised. The agreement algorithms tolerate some forms of Byzantine failures. In a purely asynchronous model, it is known that consensus cannot be reached

[1] Roughly, atomic delivery means that a message is received either by all the group members or none at all.

even if only stopping failures are considered [3]. Both Reiter and Kihlstrom et al. solve this problem using Byzantine-fault detectors that provide information about nodes suspected of having failed or of being compromised. Consensus can be reached in such frameworks, although not all Byzantine failures can be detected [7].

These distributed architectures provide much stronger intrusion-tolerance guarantees than a centralized system, but the agreement protocols are complex and can cause important performance degradation. These infrastructures are reserved for very critical applications with high-integrity requirements.

In the following sections, we examine two examples of lightweight systems that support secure group-oriented applications, namely, Antigone and Enclaves. We present the architecture of the two systems and we evaluate the protocols they use to implement group-management services and secure communication.

2 Antigone

Antigone is a modular framework developed at the University of Michigan for implementing secure group communication [12]. Antigone is a successor of software for lightweight secure group communication presented in [11]. The system is intended to support high-throughput applications such as videoconferencing, and efficiency and low encryption overhead are two major concerns. Antigone provides a set of basic *micro protocols*, each implementing a simple communication or group-management function. A group application is developed by composing these various micro protocols and selecting appropriate options according to the security policy.

The architecture of an Antigone application is shown in Figure 1. A group consists of a designated session leader and a set of regular members. In addition, the Antigone system requires the presence of a trusted third party (TTP) for user authentication. The TTP is not part of the group and can be shared between several group applications. Each group member, including the session leader, and each user willing to join the group has a long-term secret key that is also known by the TTP.

2.1 Communication Services

Antigone provides multicast communication services that can be implemented either on top of a nonsecure multicast layer or by using point-to-point protocols. In the latter case, users communicate via the session leader, which relays messages to the group members. A mixed mechanism is also available, whereby users send data to the session leader in a point-to-point mode and the leader sends data to the users by using multicast.

Application messages can be encrypted using a symmetric key shared by all the group members. Antigone uses DES for this purpose. Message integrity is ensured by message-authentication codes (MAC) implemented using MD5 [18]. Both services are optional. The possibility of additionally providing origin authentication using digital signatures is mentioned in [12] but this facility is not implemented.

The possible message formats corresponding to the various options are shown in Table 1. Each message includes the identity of the sender A and a group identifier g. The symmetric key K_g is known by all the members of group g, m is the message content, and H denotes a hash function. It is not clear why double encryption is

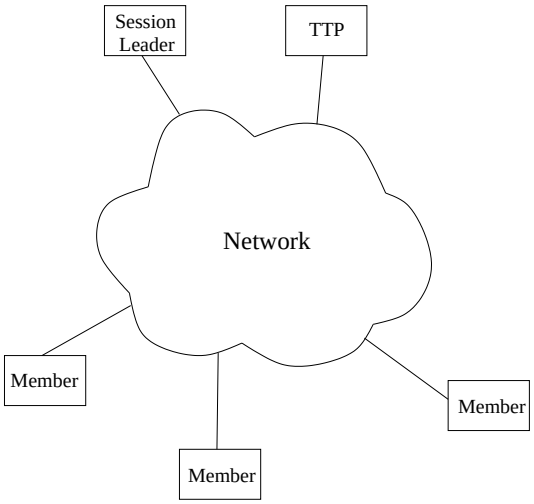

Figure 1: Antigone Architecture

Services	Message Format
Integrity	$g, A, m, \{H(g, A, m)\}_{K_g}$
Confidentiality	$g, \{A, m\}_{K_g}$
Both	$g, \{A, m\}_{K_g}, \{H(g, \{A, m\}_{K_g})\}_{K_g}$

Table 1: Application Messages

used when both confidentiality and integrity are required. Protection against message replay is not provided and must be handled at the application level.

Group members obtain the group key K_g and the group identifier g when they join the application. Each group key is associated with a particular group identifier g. Both this identifier and the group key can be reissued periodically or when the group composition changes, as described in the following section.

2.2 Group Management Services

Antigone provides different basic protocols for group and session management that can be combined as the application requires. These basic protocols provide the following services:

- User authentication and join protocol
- Group key and membership distribution
- Leave protocol
- Fault-detection protocol

Upon initiating an application, the session leader (SL) generates a public-key/private-key pair, denoted by (Pu_G, Pr_G). The public key is distributed to group members after authentication and is used by the fault-detection protocol. The session leader has a long-term secret key K_{SL} known by the trusted third party. To be able to join the application, a user A must also be registered with the trusted third party, and have a long-term secret key K_A.

2.2.1 Distribution of Bipartite Keys

Group communication optionally relies on a global group key K_g as discussed previously. In addition, for group-management operations, each user needs to communicate securely with the session leader. For this purpose, a user-specific encryption key is constructed and is shared by this user and the session leader. These symmetric keys are constructed using the scheme proposed by Leighton and Micali [9]. This scheme is intended to provide secret-key distribution in an efficient way that does not require public-key cryptography.

The key used by A to communicate with SL is $\sigma_{SL,A} = \{SL\}_{KA}$, that is, the identity of the session leader encrypted using A's long-term key. As previously, encryption relies on the DES algorithm. Key construction is then immediate for user A, who knows both SL and K_A. On the other hand, the session leader cannot construct $\sigma_{SL,A}$ since it does not know A's long-term key K_A. To obtain $\sigma_{SL,A}$, SL communicates with the trusted third party as described below:

$$\begin{array}{lll} 1. & SL \rightarrow TTP: & SL, A, I_1 \\ 2. & TTP \rightarrow SL: & \{\{A\}_{K_{SL}} \oplus \{SL\}_{K_A}, I_1\}_{K_{SL}}. \end{array}$$

In the first message, SL sends its identity, the identity of the user with whom it wants to communicate, and a nonce I_1. The trusted third party has access to both SL's and A's long-term keys. TTP can then generate a so-called pair key $\pi_{SL,A}$ computed as follows [2]

$$\pi_{SL,A} \quad = \quad \{A\}_{K_{SL}} \oplus \{SL\}_{K_A}.$$

This pair key and the nonce I_1 are returned to SL encrypted using SL's long-term key. On reception of message 2, SL decrypts the message and checks that the nonce is correct. SL can then obtain $\sigma_{SL,A}$ by

$$\sigma_{SL,A} \quad = \quad \pi_{SL,A} \oplus \{A\}_{K_{SL}}.$$

The most external encryption of message 2 is used to tie together the nonce and the pair key[3]. Checking that message 2 contains nonce I_1 ensures SL that the message is fresh and was generated by TTP (since nothing else knows K_{SL}).

As it stands, message 2 is not explicit enough. Even if the nonce is correct, SL has no evidence that the pair key it receives is actually $\pi_{SL,A}$. An attacker B could intercept message 1 and replace it with

$$1'. \quad B \rightarrow SL: \quad SL, B, I_1,$$

[2] $\oplus$ denotes bitwise exclusive or.

[3] This may have been introduced to fix a flaw in Antigone's predecessor [11].

which would cause TTP to send $\{\pi_{SL,B}, I_1\}_{K_{SL}}$. From this message SL would wrongly use $\pi_{SL,B} \oplus \{A\}_{K_{SL}}$ instead of $\sigma_{SL,A}$. The attacker B can disrupt the authentication and key-distribution process in a way that neither SL nor A can easily trace.

To avoid this problem, it would be prudent to include at least A's identity in the response from TTP. Since the reply is encrypted with K_{SL}, it is also superfluous to construct the pair key $\pi_{SL,A}$. TTP could simply send $\sigma_{SL,A}$, that is, $\{SL\}_{K_A}$. The whole key-exchange protocol could then be strengthened and simplified as follows:

1. $SL \rightarrow TTP:$ SL, A, I_1
2. $TTP \rightarrow SL:$ $\{\{SL\}_{K_A}, A, I_1\}_{K_{SL}}$.

Once we reach this conclusion, the Leighton-Micali approach has no advantage in the Antigone context. The same key $\sigma_{SL,A} = \{SL\}_{K_A}$ will be used for communication between A and SL as long as K_A and K_{SL} are valid. The same key will then be used in different sessions, in all the applications initiated by SL and in which A wants to participate. If compromised, $\sigma_{SL,A}$ enables an attacker to masquerade as A or eavesdrop on future sessions without knowing A's long-term key. A better approach would be for the trusted third party to generate a random $\sigma_{SL,A}$ for each session and communicate it to A and SL using standard key-exchange protocols. In this way, disclosure or loss of a session key does not compromise other sessions.

2.2.2 Authentication and Join

The user authentication protocol used by Antigone requires SL to obtain $\sigma_{SL,A}$ from the trusted third party as described previously. Assuming SL is in possession of this key, the protocol is as follows:

1. $A \rightarrow SL:$ A, I_0
2. $SL \rightarrow A:$ $SL, A, \{g, A, I_0, I_2, \ldots, Pu_G\}_{\sigma_{SL,A}}$
3. $A \rightarrow SL:$ $A, \{A, I_2\}_{\sigma_{SL,A}}$.

In the description given in [12], the exchange between SL and TTP to obtain $\sigma_{SL,A}$ takes place between messages 1 and 2. A initiates the protocol by sending its identity to SL together with a nonce I_0. In message 2, SL transmits to A the current group identifier g and the group public key Pu_G, together with a fresh nonce I_2 and other information about the application, such as the security policy used. On reception of this message, A can check that it is fresh (since it contains I_0) and was generated by SL (since it is encrypted using $\sigma_{SL,A}$). A then acknowledges reception with message 3. Receiving this message is evidence to S that A is now in possession of g and Pu_G.[4] After message 3, SL sends to A the current group key as described below. Optionally and in accordance with the application policy, SL can also transmit to A the identity of all the group members.

Access control can be implemented during the authentication and join procedure. For example, SL can check whether A is authorized to participate in the application (for example, using an access control list) before sending message 2 and send an error message otherwise.

[4] In [12], authentication (messages 1-2) and join request (message 3) are considered as separate activities.

2.2.3 Group-Key Distribution

As mentioned previously, all the members of a group g share a common symmetric key K_g. The key and group identifier can also be periodically changed and reissued. All these operations are performed by the session leader, and use the keys of the form $\sigma_{SL,A}$ that SL obtains during user authentication. With each key-distribution message, the session leader attaches a sequence number S_{SL} used by the fault-detection protocol. Optionally, group-key distribution messages can also contain the identity of the current group members.

Different key distribution policies are supported, which define when new group keys should be generated and distributed. Key generation can be membership-sensitive (i.e., a new key is generated when a new user joins or a member leaves the group) or time-sensitive (i.e., the group key is valid only for a limited period, after which a new key is generated).

The simplest case is when a new group member A is sent the current group key K_g just after joining the application. SL simply sends the following message to A:

$$g, S_{SL}, A, \{g, K_g\}_{\sigma_{SL,A}}, \{H(g, S_{SL}, A, \{g, K_g\}_{\sigma_{SL,A}})\}_{K_g}.$$

The current session key and the current group identifier are encrypted with $\sigma_{SL,A}$. A digest is appended and encrypted with K_g to ensure integrity of the message. Group membership information can also be included in the first message SL sends to A after authentication. In such a case, SL must also announce to the group that A has joined the session, but this is not described in [12].

The identifier g is the concatenation of a character string identifying the application and of a nonce regenerated each time a new group key is issued. It is not clear why the first component of the message is g since g is already known to A. A received the group identifier during the authentication protocol. On the other hand, including g in the third message component is critical. A must check that this g is identical to the g it received at the end of the authentication protocol. This provides evidence that K_g is actually the current group key, and to some extent that the message is recent. [5] It is important that A uses the value of g it received during authentication and obviously not the first g that is part of the message itself. If this check succeeds, A has reasonable evidence that K_g is the current group key. On the other hand, the sequence number S_{SL} may not be sufficiently protected. A has no evidence that S_{SL} originates from SL and has not been tampered with. Since the digest is encrypted with K_g, anybody who knows K_g can modify SL and change the MAC.

The above case describes the distribution of the current group key to a new member. When rekeying is required, the session leader generates a new group identity g' and a new group key $K_{g'}$. It then multicasts the following message:

$$g', S_{SL}, (A, \{g', K_{g'}\}_{\sigma_{SL,A}}), (B, \{g', K_{g'}\}_{\sigma_{SL,B}}), \ldots, \{H(g', S_{SL}, \ldots\}_{K_{g'}}.$$

This message is the concatenation of blocks of the form $A, \{g', K_{g'}\}_{\sigma_{SL,A}}$, one for each member of the group. It also includes the new group identifier g', the sequence

[5] A priori, A has no strong evidence that the message is fresh since A does not know for how long g has been in use. However, knowledge of the application rekeying policy can increase A's confidence that the message is fresh.

number S_{SL}, and a MAC produced using $K_{g'}$. This message implicitly distributes the group composition to all the members.

In this message, the recipients do not have any evidence of freshness. The group members have no way to tell whether g' is a newly generated group identifier or whether the whole message is a replay of a previous key distribution message, even a message from a past session. This is a serious threat: if B has participated in a previous session with the same leader and has kept an old key-distribution message for a key $K_{g'}$, then B can replay this old message and convince a group member to use $K_{g'}$ in an open session. This may enable B to eavesdrop on new sessions without authorization to participate. As previously, and for the same reasons, S_{SL} may not be sufficiently protected.

2.2.4 Leave and Eject

A group member wanting to leave a session simply sends the following message to the session leader:

$$A, \{g, A, \{g, A\}_{K_g}\}_{\sigma_{SL,A}}.$$

The session leader can safely assume that the encrypted part of the message originated from A (since nobody else knows $\sigma_{SL,A}$) and the presence of the group indicator g is, to some extent, evidence of freshness. [6] The block $\{g, A\}_{K_g}$ is redundant in this case.

Antigone also allows any group member to request the ejection of another member. If A would like B to be expelled from the group, it uses the same message format as above and sends

$$A, \{g, A, \{g, B\}_{K_g}\}_{\sigma_{SL,A}}$$

to the session leader. The block $\{g, B\}_{K_g}$ contains the identity of the user A wants to be expelled. It is not clear why this block is encrypted with the group key K_g; a message of the form $\{g, A, B\}_{\sigma_{SL,A}}$ should be sufficient.

The actions of the session leader after reception of such a leave or eject message is dependent on the application's security policy. After a leave request, SL may have to generate a new group key $K_{g'}$ and a new group identifier and distribute them to the remaining group members. The security policy must also specify the conditions for an ejection request to be accepted by SL and the actions to perform in such a case. In Antigone, it is not mandatory for a new group key to be installed whenever user A leaves or is ejected. However, even after A is no longer part of the group, A can still listen to group traffic and even broadcast to the group members until K_g is changed.

2.2.5 Fault Detection

Antigone supports a simple fault-detection mechanism based on *secure heartbeat* messages. This mechanism detects fail-stop failures of group members and is also intended to ensure that each member is in possession of the current group key and group membership information. There are two distinct fault-detection protocols. One allows members to detect failures of the session leader and the other is symmetrical: it allows the session leader to detect failures of the members.

[6] With the same proviso as previously: how often g changes is application and policy dependent.

Detection that the session leader has stopped uses the public- key/private-key pair that SL generates when initiating the session. The private key Pr_G that only SL knows is used to sign heartbeat messages. The public key Pu_G is sent to the group members when they join the session. The protocol also uses the sequence numbers S_{SL} included in the group-key distribution messages. Periodically, the session leader multicasts the following heartbeat message to all members:

$$g, SL, S_{SL}, H(g, S_{SL})_{Pr_G}.$$

This message contains the current group identifier g, the session leader's identity SL, and the most recent sequence number S_{SL}. A signature computed using the hash function H and the group private key Pr_G is appended. Using this private key, rather than the bipartite keys $\sigma_{SL,A}$ that SL shares with each member, reduces the number of messages SL must sign and distribute.

The digital signature provides evidence that the message originated from the session leader. However, a group member receiving such a heartbeat message has no strong evidence that the message is fresh. The same heartbeat message is produced by the session leader as long as g and S_{SL} are in use. The Antigone description in [12] does not specify clearly when S_{SL} is changed and how members are aware that it has changed. It seems that the only way a group member can know that g and S_{SL} have changed is when it receives a group-key distribution message (i.e., when it joins the group for the first time or when rekeying occurs). As already noted, the key distribution protocol is susceptible to replay attacks, and the sequence number S_{SL} attached to key distribution messages is not sufficiently protected.

Even if these two weaknesses are not exploited, an attacker can convince a member A to continue to use a compromised group key K_g by preventing the delivery of rekeying messages and replaying old heartbeat messages. As discussed in [12], one goal of the protocol is to protect against such an attack. So the fault-detection protocol fails to achieve one of its goals. For similar reasons, it also fails to achieve its other goal, namely, detecting that SL has crashed.

The protocol could be improved somewhat by requiring that members reject heartbeat messages that contain a sequence number S_{SL} they have seen before. This can be implemented by having SL increment S_{SL} with every heartbeat, and this may very well be how Antigone actually works. However, this approach does not really solve the problem; it only makes the attacks more difficult. On receiving a heartbeat message, A still has no strong evidence that the message is fresh. An attacker can convince A to continue using an old key by collecting a sufficient number of heartbeat messages and replaying them in sequence later on.

A similar approach that suffers from the same weaknesses is used for detecting crash failures of group members. Each member A is required to periodically send a heartbeat message of the following form to the session leader

$$g, A, S_A, H(g, S_A)_{\sigma_{SL,A}}.$$

S_A is a sequence number maintained by A and presumably periodically incremented. The MAC constructed using H and $\sigma_{SL,A}$ guarantees that the message originated from A, but there is still no good freshness indicator in the message.

Part of the fault-tolerance protocol includes provision for members to request the current group key. A group member A that wants to obtain the most recent group key sends the following message to SL:

$$g, A.$$

A simply sends its identity and the group identifier it currently uses in the clear. On reception, SL does not know the message's origin and has no indication that the message is fresh. Including g does not seem useful since A cannot be certain that g is the current group identifier (otherwise A would not request a key retransmission). The session leader answers to such key-retransmit requests by resending the current group key and group identifier to A, as described previously.

2.3 Summary

As can be seen from the preceding comments, the Antigone protocols have various design flaws that can be exploited by outsiders or by malevolent group members. Antigone's designers seem to have focused to a large extent on performance issues and to have forgotten the basic design principles of cryptographic protocols (e.g., as given in [1, 2]). The main problems are related to the absence of good freshness indicators in many messages, which enables various forms of replay attacks. Furthermore, the recipients of certain group-management messages cannot establish their origin with certainty, and the choice of the Leighton-Micali approach to key management is questionable.

The overall design principles of Antigone – to provide a collection of micro protocols that can be assembled as the application requires – is interesting. Each micro protocol corresponds to the various group-management operations described previously. A potential weakness is that the exact goal of each micro protocol is not clearly specified. As they stand, the micro protocols have weaknesses but even if this was not the case, one must also examine whether interactions between micro protocols can compromise security. We have not explored this aspect.

3 Enclaves

Enclaves[7] is a platform for building secure group-oriented applications over insecure networks such as the Internet. Enclaves is designed to support applications in which a set of users collaborate by performing operations on shared objects such as files, and communicate using either point-to-point channels or multicast.

A first version of Enclaves (Enclaves 1.0) was developed in 1996 [4]. A second version (Enclaves 2.0) was later implemented in Java using JDK 1.02 [6]. Significant differences exist between the group-management protocols used by the two versions, and our description is based on Enclaves 2.0 as described in [6] and on the Java source code.

Enclaves is a lightweight system that relies only on common TCP/IP protocols and provides best-effort communication services. Enclaves does not use IP multicast but only point-to-point protocols. The general architecture of an Enclaves 2.0 application

[7] Enclaves is a trademark of SRI International.

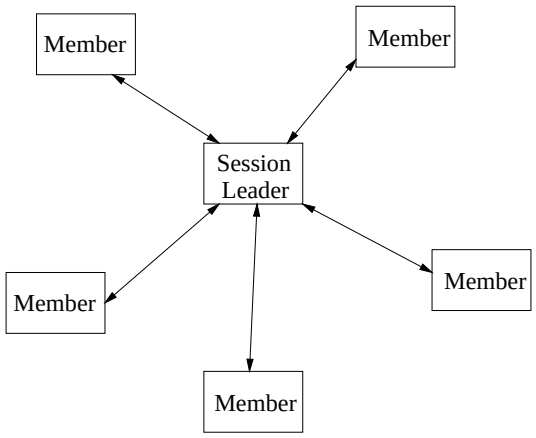

Figure 2: Enclaves 2.0 Architecture

is shown in Figure 2. As in Antigone, the group is organized around a group leader that starts and ends the application, and is responsible for all group-management actions. The other group members establish a bidirectional point-to-point link with the leader when they join the application. An important difference with Antigone is that no trusted third party is present.

Because Enclaves does not rely on any underlying multicast mechanism, all group communications are mediated by the group leader. Messages from a group member that are destined for the group must be sent to the group leader for relay to the others.

Enclaves 2.0 can support group applications implemented as Java applets running in web browsers at each member site. In such cases, the application must conform to various security restrictions imposed by existing web browsers. Typically, an applet is not allowed to communicate with arbitrary sites but only with the site from which it was downloaded. As a result, applet-based applications are organized in a star topology as show in Figure 2. The group leader is at the center of the star and must run on the site from which members download the Enclaves applets. In such topologies, direct point-to-point communication between members other than the leader is not possible. Enclaves 1.0 is less restrictive and users can establish direct point-to-point communication links. The establishment of such a link is mediated by the group leader but, once the link established, two users can communicate without leader intervention.

3.1 Communication Services

As with Antigone, Enclaves provides multicast communication services implemented using point-to-point TCP/IP protocols. In addition, Enclaves also supports a "gossip" mechanism allowing a group member to securely send messages to some other members. Thus two or more users can communicate separately from the rest of the group.

As far as the communication policy is concerned, Enclaves is not as flexible as

Antigone. All the messages, whether multicast to the whole group or sent to a subset of the members, are fully encrypted to ensure confidentiality. There is no option for providing only integrity guarantees or for sending messages in the clear. The encryption uses DES in CBC mode [6].

Messages destined for the whole group are encrypted using a group-level session key K_g. This key is distributed to new members during the authentication process as described in the following section. The session key can be changed at will by the group leader. For example, the leader can generate a new session key when a member leaves or is expelled, when a new member joins the group, or on a periodic basis. The rekeying policy is application dependent. By default, the group key is generated when the first member joins the group and remains in use for the whole session.

In addition to the group key, which is common to all the members, each member A receives a key K_a when joining the group. This key is used for group-management services and for gossiping. The approach is very similar to Antigone: members have access to a group key K_g and to a secret key K_a they share with the group leader. In Enclaves, the key K_a is valid for only one session, unlike the key $\sigma_{SL,A}$ that plays the same role in Antigone. This provides stronger security: compromise of an old session key K_a does not enable an attacker to access new sessions.

3.1.1 Multicast

To send a message to the whole group, a member A simply encrypts it with the group key K_g and sends it to the group leader L. The latter forwards the message to the rest of the group:

$$\begin{array}{lll} 1. & A \rightarrow L: & A, \{data\}_{K_g} \\ 2. & L \rightarrow B_1, \ldots, B_n: & A, \{data\}_{K_g}. \end{array}$$

An individual copy of message 2 is sent to all the group members $B_1, \ldots, B_n$. Like all other messages in Enclave, message 2 is labeled with the identity A of the originator that is sent unencrypted.

This basic protocol ensures confidentiality. Only users in possession of K_g can read the message. Encryption also gives message integrity guarantees provided the message content (i.e., the *data* field) is not random. There is no protection against replay: the leader L and the recipients $B_1, \ldots, B_n$ do not have guarantees that the message is fresh. There are also no guarantees that the message actually originated from A since any other group member can forge such a message. Such issues must be handled at the application level.

Enclave 1.0 used a much more conservative approach by including the sender's identity, together with a timestamp and a sequence number in all messages [4].

3.1.2 Gossip Protocol

To send a message to a subset $B_1, \ldots, B_m$ of the group members, A uses the following protocol:

$$\begin{array}{lll} 1. & A \rightarrow L: & A, \{B_1, \ldots, B_m,\ data\}_{K_a} \\ 2. & L \rightarrow B_i: & A, \{B_i, K'\}_{K_{b_i}} \{data\}_{K'}. \end{array}$$

In message 1, A attaches the list of intended recipients to the data and encrypts the result with its key K_a. On reception of the message, the leader extracts the list of

recipients and the data to transmit. L generates a fresh key K' and encrypts the data with K'. L then sends this encrypted data to all the recipients $B_1, \ldots, B_m$, accompanied with the key K' encrypted with each recipient's session key K_{B_i}. The key K' is used only once, for this specific gossip message. [8]

As previously, there is little protection against replay. Also, the gossip protocol requires significant computation on the leader's part: L needs to decrypt the first message, generate a new key, encrypt the data with this new key, and encrypt the new key m times. If many gossip exchanges take place, this might cause a performance penalty. By modifying the protocol, it is possible to have the sender A perform part of the work, namely, encrypt the data directly with the gossip key K'. This is possible, for example, if each member has the ability to request from the leader such a gossip key when necessary or to generate a gossip key independently and send it to the leader. With such an approach, the leader's workload is reduced: it only has to relay the encrypted data and distribute the gossip key to the intended recipients.

3.2 Group Management Services

The Enclaves infrastructure assumes that each potential group member has a long-term identification token such as a password, or can be identified using a public-key certificate. In the current implementation, only password-based authentication is supported. User passwords must be known in advance to the group leader, but password distribution is not part of Enclaves. The communication of user passwords to the leader must be prearranged before the application starts. This is different from Antigone, where a trusted third party, not the session leader, is in possession of users' long-term keys.

A group application is started by the group leader who also performs all the group-management operations. Enclaves provides the same types of group-management services as Antigone, with the exception of the fault-detection protocols.

3.2.1 Authentication and Join

Once the group leader is running, a user A can join the application. The procedure is decomposed in two steps. First, A signals to the leader its intention to join the group. The leader can either accept or deny access to A depending on the application security policy. Second, an authentication and key exchange protocol is executed. If the protocol terminates successfully, A gets the current group key K_g and a session key K_a that A will used for group-management services until A leaves the session.

The pre-authentication exchange is as follows:

1. $A \rightarrow L$: A, req_open
2. $L \rightarrow A$: L, ack_open (or $connection\_denied$)

The user simply sends an unencrypted request to join and the leader replies by accepting or refusing the connection. If the connection is accepted, A initiates the

[8] This description is roughly based on the Enclaves 2.0 documentation [6], but the actual code does not implement the gossip protocol properly.

following authentication protocol.

$$\begin{array}{lll} 1. & A \rightarrow L: & A, \{A, L, N_1\}_{P_a} \\ 2. & L \rightarrow A: & L, \{L, A, N_1, N_2, K_a, IV, K_g\}_{P_a} \\ 3. & A \rightarrow L: & A, \{N_2\}_{K_a} \end{array}$$

In message 1, A encrypts a triple composed of A's identity, the leader's identity and a nonce N_1 and sends the result to the leader. This encryption uses a key P_a derived from A's password, so P_a is known by A and by the leader. On reception of this message, L checks that the two encrypted identities are correct and extracts N_1. L then uses a random generator to produce a new nonce N_2, a new shared key K_a, and an initialization vector IV. L sends all these components together with N_1 and the current group key K_g, all encrypted with P. On reception of message 2, A checks the sender's and recipient's identities and nonce N_1. If these components are correct, A knows that the message is fresh, came from the leader, and is destined for A. Message 3 is an acknowledgment from A that it has received message 2 and then knows both K_a and K_g, and the initialization vector.

At the end of this protocol, A is ready to participate in group activities. The leader informs all the group that a new member A has joined and conversely sends to A the identity of all the other group members. These messages are encrypted using the group key. To a previous member B, L sends

$$1. \quad L \rightarrow B: \quad L, mem\_added, \{A\}_{K_g},$$

and to A, L sends m messages of the following form, one for each group member B_i,

$$1. \quad L \rightarrow A: \quad L, mem\_added, \{B_i\}_{K_g}.$$

The authentication protocol seems very well designed but it is not clear why the pre-authentication exchange is used. It may seem economical to check whether A is allowed to join before performing the authentication protocol, but A has no strong guarantee that the reply *ack_open* or *connection_denied* actually came from the group leader. This may enable a form of denial-of-service attack: to prevent a legitimate user A from joining the group, an attacker can forge a *connection_denied* reply and send it to A.

There are also some weaknesses in the distribution of group-membership information. First, sending m messages to A each containing the identity of a single member is not very efficient. It would be more economical to send a single message containing the identities of all the members.

Although this may not lead to a very serious security failure, the only indication that the messages are fresh is in the key K_g. On reception of the message that A has joined, B does not know with certainty that the message is not a replay, unless B knows that the key K_g is recent.

3.2.2 Group-Key Management

The group leader generates a first group key K_g when the first member is accepted. This key is used by the whole group,and a new group key can be redistributed at any

time by the session leader. Generation of a new group key depends on the security policy of the application. A new key can be generated when new members join, when members leave, or on a periodic basis.

For distributing the new group key K'_g, the leader sends an individual message to all the members and waits for an acknowledgment:

$$\begin{array}{lll} 1. & L \rightarrow A: & L, new\_key, \{K'_g, I.V.\}_{K_a} \\ 2. & A \rightarrow L: & A, new\_key\_ack, \{K'_g\}_{K'_g}. \end{array}$$

This is not very secure since A has no evidence that the first message is fresh. An attacker can potentially send to A an old key K'_g by replaying an earlier key distribution message. The replayed message can even be from a preceding session or a different application.

3.2.3 Leave and Expel

A user A willing to leave the session simply sends the following message:

$$1. \quad A \rightarrow L: \quad A, req\_close$$

The leader then sends an acknowledgment to A and informs the rest of the group that A has left:

$$\begin{array}{lll} 2. & L \rightarrow A: & L, close\_connection \\ 3. & L \rightarrow B_i: & L, mem\_removed, \{A\}_{K_g}. \end{array}$$

The problem here is that L has no indication that the first request actually originated from A. Anybody who can forge such a request can illegitimately remove A from the group. There is also no good freshness indicator in message 3.

By a very similar mechanism, the group leader can expel A from the group by simply closing the connection:

$$\begin{array}{lll} 1. & L \rightarrow A: & L, close\_connection \\ 2. & L \rightarrow B_i: & L, mem\_removed, \{A\}_{K_g}. \end{array}$$

As above, there is no evidence that the first message originated from the leader and is not forged. Anybody can then convince A that A has been expelled.

Although Enclaves does not make it mandatory, it would be prudent to change the group key K_g when a member is removed from the group. As long as K_g is in use, A may still be able to listen to or disrupt group communication.

3.3 Summary

Enclaves 2.0 was developed as a quick demonstration prototype and suffers from various security flaws. A much more secure and conservative approach was used in Enclaves 1.0, where all encrypted messages contained the identities of the sender and of the intended recipient, a timestamp, and a sequence number. Most of the flaws we identified in the group-management services of Enclaves 2.0 would be fixed by resorting to the same approach.

Compared to Antigone, Enclaves provides similar group-management services but in a less flexible way. The general communication facilities are also similar although Enclaves does not include services for ensuring communication integrity without secrecy. On the other hand, Enclaves provides an interesting gossip feature not present in Antigone.

4 Conclusion

Antigone and Enclaves are both lightweight infrastructures for developing secure distributed group applications. Their similar architectures are based on a central group leader, and both have the same general approach to encryption and group management. Users first join the application by authenticating themselves to the group leader. As a result, users obtain a user-specific key that they share with the leader and a group-specific session key that they share with the rest of the group. User-specific keys are used for administrative and group-management purposes that are all handled by the group leader. Group keys are used for securely communicating within the group.

Antigone and Enclaves provide different levels of protection against intrusions or attacks from different origins. The primary line of defense is an authentication protocol intended to protect against outsiders's attacks: ideally, only registered users (i.e., those who have the required long-term key or password) should be able to successfully perform the authentication. A second line of defense relies on access-control mechanisms to decide which registered users are allowed to enter the group once they have been authenticated. Once in the group, users are trusted: they can freely participate in all group communication and access any messages sent to the group. A limited form of protection against misbehaving insiders is available in the form of an expel mechanism. Neither Enclaves nor Antigone has any built-in way of detecting that a user misbehaves: the decision to expel a particular user is left to the application. Finally, both Enclaves and Antigone rely on a central, totally trusted group leader. There is almost no protection against a nontrustworthy leader. Antigone just provides a fault-detection mechanism to signal that the leader has crashed.

Unfortunately, both Antigone and Enclaves 2.0 suffer from various security flaws that enable various forms of intrusions. Most of these flaws are due to insufficient evidence of freshness and proof of origin of the messages used in many group-management operations.

References

[1] M. Abadi and R. Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, January 1996.

[2] R. Anderson and R. Needham. Robustness Principles for Public Key Protocols. In *Proceedings of the International Conference on Advances in Cryptology (CRYPTO'95)*, volume 963 of *Lecture Notes in Computer Science*, pages 236–247. Springer-Verlag, 1995.

[3] M. Fischer, N. Lynch, and Paterson S. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, April 1985.

[4] L. Gong. Enclaves: Enabling Secure Collaboration over the Internet. In *Proceedings of the 6th USENIX Security Symposium*, pages 149–159, San Jose, CA, July 1996.

[5] L. Gong. Enclaves: Enabling Secure Collaboration over the Internet. *IEEE Journal of Selected Areas in Communications*, 15(3):567–575, April 1997.

[6] S. Keung and L. Gong. Enclaves in Java: APIs and Implementations. Technical Report SRI-CSL-96-07, Computer Science Laboratory, SRI International, Menlo Park, CA 94025, 1996.

[7] K. Kihlstrom, L. Moser, and P. Melliar-Smith. Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector. In *Proceedings of the International Conference on Principles of Distributed Systems*, pages 61–75, Chantilly, France, December 1997.

[8] K. Kihlstrom, L. Moser, and P. Melliar-Smith. The SecureRing Protocols for Securing Group Communication. In *Proceedings of the IEEE 31st Hawaii International Conference on System Sciences*, volume 3, pages 317–326, Kona, Hawaii, January 1998.

[9] T. Leighton and S. Micali. Secret-Key Agreement without Public-Key Cryptography. In *CRYPTO'93*, pages 456–479, Santa Barbara, CA, August 1993. Springer-Verlag, LNCS 773.

[10] B. Levine and J.J. Garcia-Luna-Aceves. A Comparison of Reliable Multicast Protocols. *ACM Multimedia Systems Journal*, 6(5):334–348, September 1998.

[11] P. McDaniel, P. Honeyman, and A. Prakash. Lightweight Secure Group Communication. CITI Technical Report 98-2, Center for Information Technology Integration, University of Michigan, Ann Arbor, MI, April 1998.

[12] P. McDaniel, A. Prakash, and P. Honeyman. Antigone: A Flexible Framework for Secure Group Communication. In *Proceedings of the 8th USENIX Security Symposium*, pages 99–114, Washington, DC, August 1999.

[13] K. Obraczka. Multicast Transport Protocols: A Survey and Taxonomy. *IEEE Communications Magazine*, 36(1):94–102, January 1998.

[14] M. Reiter. Secure Agreement Protocols: Reliable and Atomic Group Multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communication Security*, pages 68–80, Fairfax, VA, November 1994.

[15] M. Reiter. A Secure Group Membership Protocol. In *Proceedings of the IEEE Symposium on Research in Security and Privacy*, pages 176–189, Oakland, CA, May 1994.

[16] M. Reiter. The Rampart Toolkit for Building High-Integrity Services. In *Theory and Practice in Distributed Systems*, pages 99–110. Springer Verlag, LNCS 938, 1995.

[17] M. Reiter. A Secure Group Membership Protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, January 1996.

[18] R. Rivest. The MD5 Message Digest Algorithm. Internet Engineering Task Force, RFC 1321, April 1992.