Security for Network Places

Tim Kindberg

Dept. of Computer Science, Queen Mary & Westfield College, University of London
London E1 4NS.
timk@dcs.qmw.ac.uk

1. Introduction

The Mushroom project[1] is developing a framework to support network 'places' for wide-area collaboration and interaction [1-4]. A network place is a persistent environment in which a collection of users can share information objects and communicate with one another. We envisage a world consisting of many places, each supporting a group of users performing a specific task or closely related set of tasks. For example, a group of authors spread around the world can use a network place to manage drafts of a book on which they are collaborating; a software company can use a place to hold sources and design documents for each piece of software that its geographically distributed programmers are developing.

Much of the information that users store in places and their communications within them are confidential: that is, users require that they be kept secret. Users also require that the integrity of their information is maintained. We are currently investigating the use of network places to support diabetic patient care, where it is essential that both the confidentiality and integrity of patient records are maintained. Similarly, companies need their workers to collaborate without fear of eavesdropping by their competitors, or fear of working with corrupted or inconsistent information. To achieve these goals, users apply security and integrity policies which allow working only with users they can trust, and which take into account the degree of trust. Collaborators require the system to enforce their secrecy and integrity policies against attack, and also to provide an audit trail in case security is breached.

This paper discusses the issue of security for network places, and outlines our approach. Under the heading of security, we include both maintaining the secrecy of stored and transmitted information, and maintaining its integrity[2]. There are two key questions which our project is exploring:

How to provide an access control model for naive computer users such as doctors, who require role-based access control [5]. Existing security models for group working are often so complex that even their designers have trouble elaborating them [6]. Ellis et al [7] state: "Changing an object's access permissions should, for example, be as easy as dragging the object from one container to another." This paper describes how we have taken up that challenge. First, each place is a container: it has a reference monitor which uniformly controls a user's access rights with respect to all objects within it. Rights are granted not only to the generic operations that the user may perform on the objects, but also to the form in which the objects appear to the user. For example, a researcher is allowed only to read patient-record objects, and he sees the information in anonymised form (i.e. with all identifying information removed). Second, users can constrain imports and exports of objects to and from the place. We call the object that exercises import and export control a *boundary object*.

How to protect the past. Records of past events and object states are often of interest in collaborative activity. First, users need to be able to undo actions. Second, they review object histories and derive alternative objects from previous versions. Third, users need to be able to work disconnectedly, storing the operations or intermediate versions, and merging their work on reconnection. Mushroom stores previous versions of some objects. Moreover, it stores updates, in the form of objects called *events*, which Mushroom uses to propagate changes. Protecting historic data is an important issue with some subtle ramifications. Consider a doctor who writes a comment into a patient's record, and subsequently deletes it in order not to reveal it to a newly-joined clinician. The new clinician must not have access to the object's history, if the comment is to remain secret, for it will contain the event that described the original comment.

In Section 2 we describe and motivate the features of the Mushroom system needed to understand the security requirements. Section 3 describes the access control model and discusses the issues raised, particularly with respect to securing events. We conclude with a summary and outline of the open issues.

---

[1]UK EPSRC grants GR/L14602 (1996-97), and GR/L64300 (1998-2000).

[2]Integrity is not only a security matter: it may be breached by friendly accesses, for example in the absence of adequate concurrency control. But we concentrate here on security.

2. The Mushroom system

Users perform their work by sharing objects in places. They either work synchronously (in real-time, often while communicating over an audio or video link), or asynchronously (at different times) and even disconnectedly. The shared objects are written in Java and we shall refer to them as applets[3]. A simple applet is either a wrapper for a legacy document such as a word processing file or a web page, or a group-aware tool such as an electronic whiteboard or patient record. Other applets provide links to places. In particular, a place that is reachable only via one other place is called a nested place. At the system level, each place is an object which maintains a directory of the applets it contains.

For reasons of security or otherwise, each object (place or applet) may exist in several *flavours*. Objects of different flavours vary in their behaviour or state but perform related functions. Developers implement flavours either using several classes or using a single class that may function as any particular flavour. An instance of a particular flavour is called an *alternate* of the object[4]. For example, clinician A may view and keep a local copy of a patient's record in its entirety, whereas clinician B may only use an alternate which does not contain certain sensitive information, or on which certain operations are not possible. The clinicians think of themselves as sharing a single object (although they may realise that it appears differently to each), but in reality each has his or her own alternate.

Updates are communicated in Mushroom as events, which are delivered wherever an object has subscribed to them. We say that an object *applies* an event that it handles. An event [8] is an object which describes a state of affairs, for example "Sid added the paragraph p at position x in document d", encoded as a set of values. Events differ from object invocation messages in that they do not necessarily designate a particular target object, and they are first-class objects which can be stored and queried.

Events allow us to manage heterogeneous alternates of the same applet. The alternates of the same applet can raise and respond (differently, if need be) to a common set of events in order to make their state consistent. Another advantage of events is that they provide a convenient way to schedule operations upon alternates. If we order event-delivery according to the consistency constraints that apply in the application, then workstations and servers need only apply concurrency control locally to ensure consistent updates [9]. Finally, since events are first-class objects, we can construct services which process them on behalf of applications. For example, services can log and replay them [12].

Places require the following system support:

An active store. Persistent and replicated storage is needed for the place, its applets' state, and events, to allow asynchronous and disconnected working. We may replicate alternates between several servers and client sites, in order to achieve high availability. Thus what appears to the users as a single object is represented by alternates of several flavours, each of which may be one of several replicas. The store is active in that it must produce these different alternates as clients require them. Another reason why the store needs to be active is that object histories need to be *agreed*. Users can make incompatible changes to an object when working concurrently. To support sharing, we must define what is an alternate of the same shared object, as opposed to what must be regarded as a derived version of the object with a separate identity. Agreement defines the set of events that belong to any legitimate alternate of the object, and any ordering constraints that apply. Agreement is sometimes carried out at a single site (typically a server), but it need not be so. For example, contributions to a 'chat' applet could be agreed by default. And the agreed ordering is not necessarily total: some applets have more relaxed consistency constraints.

Shared channels for audio and video communication and events. Both media and events need to be pushed on a best-effort basis to users who are collaborating in real-time, in order that they can interact. These channels are manufactured and destroyed as they are required.

3. Securing places

Securing a place amounts to securing its active store and the event and media channels. Our design goals are:

- **Providing integrity and confidentiality guarantees for individual places**. Security applies not only to the state of objects and media in the place, but also with respect to what are the agreed events, and who may access them. Only authenticated users who are *members* of the place are to be able to access the objects and communicate within it. Collaboration inside the place requires role-based access control [14]. Users must be assured that they are acquiring bona fide data.
- **Import and export controls**. During the course of their work, users need to import objects into a place, and to export objects or copies of them to other places. This requires more control than blanket *create*

[3]Our applets are not the same as instances of the Java Applet class, but are similar in terms of being downloadable active content.

[4]An alternate is similar to the notion of view in the database world, but in CSCW 'view' typically means a user-interface representation of a shared object.

and *delete* rights. The system should support policies which prevent the import of objects that threaten the overall integrity of the place's objects; and it should be possible to constrain the export of information to other places (which may have different security attributes).

- **Supporting dynamic groups of users**. We allow both the members of a place and their roles within it to change. An important problem is how to maintain the consistency of a place's security attributes. We also need to support restrictions on access to historical data.
- **User-friendly operation**. This is not just a convenience. A framework that makes security policies difficult to implement, and security attributes difficult to keep track of, is liable to lead to mistakes and therefore vulnerability to security attacks.

Access control model

We are investigating a container-based access control model for places [15], in which access is controlled via a per-place access control list (ACL), rather than per-object ACLs. We do so on the assumption that per-place ACLs are more tractable for users such as clinicians. Since the security attributes of an object are a function of its location, a place has a *boundary object*, with methods *import* to bring an object into the place and *export* to move the object or a copy of it to another place. (Of course, a boundary may refuse an export or import altogether. But we cannot secure a place against leaking by one of the collaborators.) A place's *owners* determine its ACL. Recall that places may be nested. One must be a member of the enclosing place to access a nested place. Consider a place for keeping data pertaining to a patient. Data which may be read by all clinicians is kept at the top-level place. More secure data – for example, data which is intended only for the patient and the GP to see, and not nurses or other doctors – is kept in a secure nested place.

It is easy to see how read-only data can be protected using such a scheme. Standard classifications such as "top secret", "confidential" and "unclassified" can be realised using nested places to store the respective objects. How can we give access rights to those who need to update objects? Protocols exist for users who have the same rights to shared objects [11], but users such as clinicians require different rights according to their role.

A user has uniform rights to all objects within a place, taking the form of:

(1) the flavours of the objects which the user is allowed to access (e.g. only 'anonymised' flavour), and
(2) one of a small number of generic types of access, for example *edit* (read/overwrite), *annotate* (read/add), *read*, *append* (add), *manage* (read/synthesise events) and *own* (change place's ACL).

For example, consider a place containing items from a patient record. Those with the role of nurse who are only able to enter data may have at most append rights and receive a 'data-entry' flavour (with no state). GPs and consultants may have edit or annotate rights, and may be given different flavours to respect the confidentiality of certain information.

We do not yet know whether this model will in fact suffice for the clinicians. Container-based security is inconvenient for changing the access to one of a group of objects, although by moving objects between places any combination of rights can be achieved. However, we believe that our model of places as containers with boundaries has certain advantages even if, eventually, per-object ACLs prove necessary:

- boundaries can be used to maintain the integrity of collections of objects, through their 'import' functionality. For example, a boundary could enforce the integrity constraint "data pertaining to no more than one patient may exist in place P". Additionally, boundaries provide a framework for constraining information flow out of the place.
- the model corresponds intuitively to that which is familiar to our users: papers are kept in locked filing cabinets inside locked rooms. Network places have the advantage over physical places of being able to represent to the user precisely who is able to enter each place, and what type of updates, if any, they can perform on objects in that place.
- Even if per-object ACLs were to be used, it is useful to give rights only to principals of the form (Tim **and** memberOfPlace) – that is, Tim must be a member of the place to access any object within it. One has only to remove membership to deny all access to a place and to places nested within it.

Note that this model is independent of the user-interface representation. A patient record could in principle appear in the form of a set of WWW pages which are in a collection of nested containers. When a user accesses the root page, what she sees (the alternate she retrieves) depends upon who she is and/or what role she plays. She may not see a link at all to certain information. On following a link, she may or may not be able to update the data referred to. Each 'page' (sub-object) is in a secure container, but this fact is relevant only to an owner of the root place. An owner could be given a container-like view in order to set the protection settings.

Securing alternates and events

The basic features of the security design are:

- trust only certain servers and user machines operated by certain principals to supply information; the degree of trust may vary – clinicians tend to be distrustful of servers which aggregate data [5]
- ensure that only those with read rights may read alternates (of a flavour appropriate to the requesting principal) and events

- provide secure event agreement, in particular with respect to places, so that the security attributes of a place are unambiguous for any point on its time line; and only events that do not conflict with the security attributes appropriate for the place and time will be applied by 'good' principals
- allow the owners of a place (who may set its security attributes) to bound a user's access rights so that he may obtain no alternate state or event prior to a given time
- prevent all principals from altering history – changing the set of past events or substituting saved alternate state.

The strategy to secure objects and events is:

- events and snapshots of alternate state are transmitted in encrypted form
- snapshots of alternates are signed so that the recipient can check that they were manufactured by a trusted party
- conformant sites (i.e. those that are operated by trusted parties) securely check alternate snapshots, and events before applying them to alternates.

In order to consider what it means to check the legitimacy of an event before applying it, let us first distinguish between events of different types:

update events describe an update to an individual object, such as "Tim added paragraph P to D at time T"; on application to an alternate, they are equivalent to an operation upon the object

relational events make an assertion about one or more objects, but do not imply that the announcer made an update. For example, "Tim entered place P at time T".

In the first case we need to check that Tim had the right to add the paragraph at time T. An applet maps the update events it consumes into one of the categories *append*, *annotate* etc. Each alternate checks that Tim had the corresponding rights in the place – which means that it must have its own local copy of the place's security attributes. In the case of relational events, the question is whether the announcer has (had) *jurisdiction* over the assertion made by the event. An applet such as a log which applies this event is not concerned about any rights over itself: it is concerned about whether it is recording a valid state of affairs. To support this case, a principal may be granted *manage* rights within a container. This means that that principal may read objects within the container and announce relational events. These rights imply more privilege than read rights, since others are asked to believe the accuracy of what is described. They may apply non-trivial updates in response to a relational event. For example, if two objects in a virtual world are told that they have collided with one another, then they may each respond by updating their shapes.

Security over time

We now turn to the questions of how to control access to historical data, and how to manage the evolution of security attributes over time. First, consider a history object which contains a version tree of snapshots of the development of an object, and the events that map the development. For example, a place or a document within a place could have a history. The problem of securing the past reduces to the problem of securing history objects, which are just another type of object in a place. We do not have space to do justice to this problem here, but some obvious policies that users may require are:

- No user may alter any event or snapshot once it has been stored in a history.
- No user may insert an event before an existing event in the history.
- Only trusted users may delete parts of a history, and such parts must correspond to the history's tail (so that no gaps are produced).
- Users are given access to the history object only from a certain point in time – this deals with the problem of securing events (and saved snapshots) which record confidential information that has now been deleted.

To manage the evolution of security attributes is to solve two problems:

- How can we ensure that the mappings *user* → *role(s)* → *rightsInPlace* are consistently evaluated at different sites where objects are replicated?
- How can we revoke user rights?

We propose to solve these problems by implementing a service which securely defines and announces epochs – intervals – during which a place's security attributes are unchanged. Rather than assuming accurate physical clock synchronisation, we define epochs with logical timestamps.

When the place's membership or other security attributes changes (such as when a user's rights are revoked), the secure epoch service issues an epoch certificate, which declare the new epoch number, the security attributes of the place, and a timestamp consisting of the last sequence numbers of events issued by each of the place's members in the preceding epoch. An active member of the place knows the current epoch, and expects subsequently to receive either 'new epoch' certificates (when the security attributes change) or regular 'epoch confirmation' messages, stamped with the current logical time, declaring which epoch currently applies.

A member is given access to events and snapshots starting from a specified epoch (which may be in the past). Past events are obtained, as we described above, from history objects. Sites propagate current events directly to one another as users interact. The epoch to which an event belongs can be determined unambiguously except that a new epoch certificate may be on its way, possibly rendering the event non-applicable[5]. A site can wait until an epoch certificate or epoch confirmation message arrives before applying the event. If waiting for a message from the secure epoch service makes interactive latencies unacceptably high, sites can apply events tentatively [3]. They must then be prepared to undo the event if a subsequent epoch message invalidates it. Note that, whether or not sites first apply events tentatively, this scheme orders events unambiguously with respect to the beginning of each epoch, but does not constrain their relative ordering within an epoch.

4. Conclusions

We have outlined the Mushroom model of persistent shared places, and the security requirements attaching to places for collaborative activities such as patient care. We are pursuing a container-with-boundary-based security model, which we believe to be widely applicable. Users have uniform rights to objects in the place. Boundaries enforce the secure import and export of objects and events between places. Even if per-object access control lists prove desirable (as opposed to container-based security), boundary objects still have some desirable properties. These include being able to constrain access to the objects within a place globally, and being able to manage the place's integrity.

We have shown that events raise some interesting security issues. First, they are not necessarily the same as 'operations'. Second, it is often desirable for collaborating users to have access to the past, but care must be taken over what a new member of a collaboration is allowed to access.

Finally, we have raised the issue of how to manage the evolution of security attributes. Data consistency is anyway a thorny problem, when considering how to enable distributed users to work on shared objects, sometimes disconnectedly. How can we ensure that distributed sites apply security policies consistently, while allowing relaxed consistency between (some) shared objects? The answer seems to involve ordering of security-related events, without forcing synchronisation of the system (which would be an unreasonable constraint for most users). We are currently investigating practical solutions.

References

[1] T. Kindberg (1996). Mushroom: a framework for collaboration and interaction across the Internet. Proc. CSCW & the Web, 5th ERCIM workshop, Busbach, U., Kerr, D., and Sikkel, K. (eds), GMD, pp. 43-53.

[2] T. Kindberg (1996). A stake in Cyberspace. Proc. 7th. ACM SIGOPS European Workshop, Connemara, Eire, Sept., pp. 83-88.

[3] T. Kindberg, G. Coulouris, J. Dollimore and Jyrki Heikkinen (1996). Sharing objects over the Internet: the Mushroom approach. Proc. IEEE Global Internet 1996, London, Nov., pp. 67-71.

[4] Mushroom project home page: http://www.dcs.qmw.ac.uk/research/distrib/Mushroom.

[5] R. Anderson (1996). A security policy model for clinical information systems. Proc. IEEE Symposium on Security and Privacy 1996.

[6] K. Sikkel (1997). A group-based authorization model for cooperative systems. Proc. ECSCW97, pp. 345-360.

[7] C. Ellis, S. Gibbs and Rein, G (1991). Groupware - some issues and experiences. Communications of the ACM, vol. 34, no. 1, pp. 38-58.

[8] J. Bacon, J. Bates, R. Hayton, and K. Moody (1996), Using Events to Build Distributed Applications. Proc. 7th. ACM SIGOPS European Workshop, Connemara, Eire, Sept., pp. 9-16.

[9] A. Schiper and M. Raynal (1996), From group communication to transactions in distributed systems. Comm. ACM., vol. 39, no. 4, pp. 84-87.

[10] A. Rowley and J. Dollimore (1996). Replicated secure shared objects for groupware applications. Tech. rep. 716, Dept. of Comp. Sci., Queen Mary & Westfield College, U. of London, Proc. ERSADS 97.

[11] L. Gong (1996). Enclaves: enabling secure collaboration over the Internet. Proc. USENIX Unix and network security symposium, San Jose, July.

[12] W. Edwards and E. Mynatt (1997), Timewarp: techniques for autonomous collaboration. Proc. CHI 97, Mar., pp. 218-225.

[13] D. Dean, E. Felten, and D. Wallach (1996), Java security: from HotJava to Netscape and beyond. Proc. IEEE Symp. on Security and Privacy, May.

[14] G. Coulouris, J. Dollimore and M. Roberts (1998), Secure communication in non-uniform trust environments. Submission to ECOOP Workshop on Distributed Object Security, Brussels, 1998.

[5]Events may be sent after a new epoch has begun, but before the sender received a new epoch message. Recipients simply translate the timestamps from the old epoch to the new.

[15] S. Greenwald and R. Newman-Wolfe (1994), The distributed compartment model for resource management and access control. Tech report TR94-035, dept. of Computer and Information Sciences, Univ. of Florida.