Security in a Mobile Agent System

Guido J. van 't Noordende, Frances M.T. Brazier, Andrew S. Tanenbaum
Vrije Universiteit, Amsterdam, The Netherlands
{guido,frances,ast}@cs.vu.nl

Abstract

This paper describes a security architecture for the Mansion mobile agent system. Mansion is a logical framework designed to support large-scale heterogenous mobile agent applications. Mansion is implemented as a multilayered middleware system in which the lowest layer provides functionality that is common to most mobile agent systems and higher layers become increasingly application aware. The security architecture presented in this paper provides secure agent communication, secure mobile agent transport and startup, and secure auditing of all changes made to agents. The system uses self-certifying names for authenticating principals in the system and provides mechanisms to control information flow.

1. Introduction

Mansion is a system aimed at supporting heterogenous, large-scale distributed mobile agent applications. Most current mobile multiagent systems (MASes) offer little structure to application developers. Mansion provides a clear paradigm for designing applications.

A number of advantages relating to agent mobility have been described [5, 17, 8]. The most significant of these is that an agent can move its computation to the resource or data which it needs, which alleviates problems due to latency or bandwidth limitations. Security advantages relating to using mobile code have not often been described. Our design describes such an advantage relating to information flow control.

Most existing MASes provide some degree of security to agents and machines in the system. Often, only part of the security issues (e.g., host protection) are solved, leaving open several venues for attack. Usually, security solutions are tied to a single programming language (e.g., Java) and based on protection mechanisms such as user-level sandboxing [14, 18]. Mansion supports heterogenous programming languages and attempts to address security issues throughout its design, where possible at distinct layers of the system.

Important aspects for security addressed in this paper are protection of agents, protection of hosts, protection of information, and protection of the middleware. This paper briefly introduces the Mansion paradigm and then explains how the identified security areas are addressed, followed by a discussion and related work.

2. The Mansion Paradigm

2.1. Logical Model and Design

An application in Mansion is modeled as a closed world containing a set of hyperlinked rooms. Entities in a room can be agents, objects, or hyperlinks. Each agent is a (possibly multithreaded) process running on one host. No part of the internal process state of an agent can be accessed from the outside by other agents. Objects are strictly passive: they consist of data and code hidden by an interface and cannot invoke methods on other objects.

Hyperlinks determine how the rooms in a world are connected. An agent can only be in one room at a time, but can follow hyperlinks to migrate to another room. An object may reside in only one room.

A world consists of one or more *sections*, each containing a set of rooms. A room can only be in one section. Each section has one or more *Section Entrance Rooms (SERs)*, which are the only valid entry points for a section. An agent enters the world using a specially marked SER called the *World Entrance Room.*

An agent is injected into a world by its owner using a *World Entrance Daemon (WED).* WEDs are run by the world's owner and are the only way in which a world can be entered. A WED enforces a world entrance policy.

When injecting an agent, an agent's owner selects an *Agent Management Station (AMS)* for its agent from a list provided by the WED. The AMS is trusted by the agent owner and keeps track of the agent's whereabouts, such as its physical location and other facts,

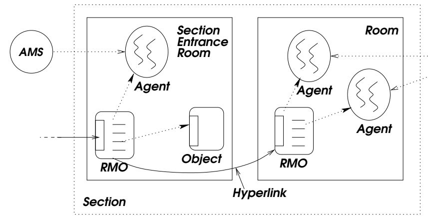

Figure 1. Overview of a section with two rooms. Two AMSes are shown which keep track of agents in the world.

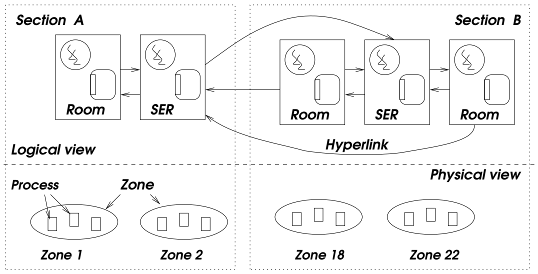

Figure 2. Logical and physical view of a world.

throughout the lifetime of the agent. AMSes are trusted by the world owner, but they need not be (and generally are not) managed directly by the world owner.

On world entrance, each agent receives a unique, world-wide identifier, *AgentID*. This identifier can be used by any agent to send messages to or open a (location-transparent) connection to the agent, irrespective of where it is in the world.

Once in an (entry) room, an agent is automatically registered in, and connected to, a special object in the room, called the *Room Monitor Object (RMO)*. The RMO registers all content in the room. All entities, by definition, have to be registered in the RMO of the room they are in. In a sense, an RMO is the core of a room, as it contains all critical data about the room and its content.

Descriptions of entities in a room (e.g., agents, objects and hyperlinks to other rooms) are specified in *Attribute Sets (ASes)* in the RMO. An attribute set is a set of attribute-value pairs. Example attributes are the name of an agent or a description of an object or a target room. Attributes are defined globally per world.

Fig. 1 shows a section with two rooms. An internal hyperlink is shown, which is essentially a pointer to an RMO. At the left, a hyperlink from a room outside the section to this section is shown. Also, two AMSes are shown which keep track of agent contact information.

2.2. Physical Distribution

A world may be spread over multiple machines. In particular, a room can be distributed by means of replication of its RMO and other objects in the room to multiple machines. Objects in Mansion can be replicated to multiple machines anywhere in the world and be accessed remotely [1]. However, in practice the physical distribution of sections, rooms and objects is constrained by administrative or security concerns.

For example, sections are spread over (generally disjoint) sets of machines, from which the rooms in this section are accessible. Each section has a *distribution policy (DP)*, set by the section administrator, which defines from which physical locations the section is reachable. Distribution policies are what maps the logical world (of disjoint sections containing rooms) to the physical world (of processes on different machines). All rooms in a section have the same DP and the same owner. An owner may create a new section for rooms that have different distribution requirements than existing sections (e.g., for security reasons).

A *zone* is a group of processes (on a set of machines) which is referred to by a single, cryptographically protected name, used to express distribution policies. Secure naming of zones is explained in detail in section 6. All zones in a world are registered centrally with the world owner and published in a global *zone list* (sec. 7). The world's zone list is known to all participants in a world; only registered zones may be used in distribution policies.

An agent has to physically migrate to one of the zones in a section before it can access a room there, if it is not in one of the section's zones yet. An agent does not have to migrate physically if it follows hyperlinks within a section, as all rooms in the section share the same DP.

Physical migration happens automatically if required when an agent follows a hyperlink. An agent can follow a hyperlink using a method *follow_hyperlink* provided by the Mansion API (sec. 3). This method atomically transfers the agent to another room, and if necessary to another physical location. If following a hyperlink fails for any reason, the agent resumes execution where it left off.

Fig. 2 shows how two sections are mapped onto zones. Sections are *logically* disjoint, but not necessarily physically (i.e., section DPs may overlap). This is useful for extending a section's DP when load increases or to increase availability, for example, with zones provided by a hosting company. These zones can be used by different sections to replicate rooms on.

2.3. Example

As an example of the Mansion paradigm, consider a shopping mall world. A shopping mall is modeled as a number of separate sections, each representing a store which contains a set of hyperlinked rooms. Each section has one or more entry rooms, which act as portals to the store. Each section is administered by the store's owner. The world owner provides a world entry room which contains hyperlinks to the different stores in the mall. In each room in the mall there may be objects that represent items (for example clipart or music) for sale, and shopkeeper agents which can be queried for information or be involved in commercial transactions. The attribute sets in the room's RMOs contain information that helps the agent find relevant items in the room.

Agents that represent users can roam through the mall to find items to their liking. Agents can communicate with each other to speed up their search or notify each other of interesting bargains. An agent may take some form of digital cash with it to be able to buy items for its owner. Items that are bought by an agent can be transported to their owner by means of inter-agent communication or as part of the agent, possibly encrypted with the public key of the agent's owner.

3. Architectural Design

3.1. Mansion Middleware

Mansion is implemented as a middleware system which provides an application programming interface (API) to agent programmers. This API is used by agents to do all their work in a world. The Mansion Middleware (MMW) acts as a *reference monitor* with regard to the agent's invocations; it takes care that agents cannot obtain access to resources or entities outside the world. For example, Mansion ensures that an agent can only use objects in its own room, and that it can only migrate to another room by following a hyperlink from this room.

The Mansion Middleware mediates access to objects. A mechanism is provided by the MMW, called *binding*, which is used to transparently connect to an object, irrespective of that object's actual physical location. An agent only sees an object's *entity-identifier (EntityID),* an index relative to the room, which it uses for binding.

Object types (interfaces) differ per application, and are predefined per world. An agent is linked (statically or at runtime) with stubs for each object type. As part of binding, the middleware connects the agent's stub

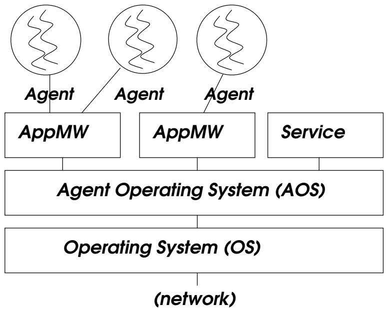

Figure 3. Middleware layering.

to the (possibly remote) object instance. The Mansion middleware hides underlying implementation and distribution aspects from agents. When possible, an object is replicated so that agents can access the object from a replica close-by (possibly on their own machine); so, agents maintain proximity to the data they use.

To provide uniform semantics when migrating, Mansion only supports *weak migration.* This means that an agent's execution state (e.g., stack and registers) are not retained when an agent migrates. In effect, an agent is killed and restarted from its initial state (as provided by the agent programmer when it was injected in the world) each time it migrates. This is in part because we want to be able to support agents in any programming language (e.g., Lisp, Java, binary agents), without modification. Also, the restart semantics makes it possible to tightly control information flow (section 9.3), something which is hard to do using strong migration.

Following a hyperlink is an atomic operation: if succesful, the agent is started up in the context of a new room, if not, the agent resumes execution where it left off with an error code indicating the reason for failure. In Mansion, an agent has to save whatever state it requires, to resume execution in a useful manner, explicitly before following a hyperlink. It can store this state in a container which is associated with the agent. This container is explained in sec. 5.

3.2. Middleware Architecture

We designed our system as a multilayered middleware system (fig. 3). The lowest layer above the operating system is called Agent Operating System (AOS). AOS's main function is as an abstraction layer which hides differences in the operating systems on which it runs from higher layers. AOS provides an interface to higher layer middleware systems with primitives for secure communication, process startup, and agent storage and migration. AOS is trusted by the layers above it.

Above AOS is a layer we call Application Middleware (AppMW), which provides middleware abstractions and services specific for a given application. The Mansion middleware (MMW) is an AppMW. Only the AppMW layer is aware of the specific concepts, policies and requirements for its application. The AOS layer does not know the entities in the application that it serves, or about trust in those entities. For example, naming services are in the AppMW layer.

AOS is currently implemented as a (user-level) 'kernel' process running on one host and serving one or more applications (AppMW processes) on the same machine. AOS and AppMW run as separate processes communicating with each other using RPC calls over a trusted IPC channel provided by the OS. AppMWs are distributed over AOSes running on multiple machines; AppMW processes interact (communicate) with each other using mechanisms provided by AOS.

In the remainder of this paper, we first introduce security mechanisms provided by the OS and AOS layers. Then we introduce self-certifying identifiers, a core concept in Mansion. After that, we discuss Mansion specific security mechanisms. We conclude with related work and a summary.

4. OS Level Security

Standard operating system protection mechanisms are not sufficient for executing mobile agents in a controlled way. For example, in Mansion, we want to ensure that an agent cannot bind to an object in a different room, or set up connections to arbitrary endpoints outside the system. Particularly, we have to make sure that an agent can only communicate (using IPC) with its middleware, so that it cannot bypass the middleware's access control mechanisms.

User-level sandboxing techniques such as provided for many type-safe languages such as Java and Safe-Tcl [18] provide mechanisms to control the actions that an agent may take. The required constriction of an agent in the way needed for Mansion is obtainable in most user-level sandboxes. However, current sandboxes have inherent vulnerabilities and may contain bugs that can put the system at risk [9, 25].

Current research indicates the need to include a 'red line', or kernel abstraction [10, 25] in user-level sandboxes, particularly the JVM. We chose to revert to OS kernel level protection mechanisms where possible; we feel that, even if we have user-level sandboxes at our disposal, additional protection is needed to protect the system from malicious (binary) agents or implementation errors which may expose the system to risk. We implemented *system call interception (jailing)* in Linux [2]. A jail catches all system calls of a (binary) child process, and reflects them to a user-level process (the *jailer*). The jailer evaluates if the system call may proceed or not based on a policy file and informs the OS kernel of its decision. A similar jailing facility exists in the FreeBSD operating system [21]. In our system, the jailer is part of AOS on those platforms where jailing exists.

5. AOS Level Security

AOS provides a secure container for storing an agent and its data, called an *Agent Container (AC)*. The AC is used for shipping an agent between AOSes running on different machines. The AC is a small, portable file system, managed by AOS, which can be shipped to any platform running AOS to start the agent and access its associated data there. The AC consists of a set of typed segments, essentially binary files. An example segment-type is *code*. Segments also have a subtype description, which can for example describe the specific code type in the segment, e.g., *Java_1.3_bytecode*. Segments may be persistent or transient. Transient segments may be changed or deleted. Persistent segments may not be modified. A transient segment can be made persistent but not the other way around.

Each AC has a table of content (TOC) with an entry per segment containing its name, type, subtype, a persistence bit, and a checksum (SHA-1 hash) of the segment's content. Checksums are generated only when an agent is migrated. AOS contains a basic AC integrity verification mechanism based on the TOC as part of the agent migration protocol with which an agent is shipped to another AOS kernel. Prior to sending an agent, the sending AOS signs the AC's TOC using its private key. The receiving AOS can now verify the AC's integrity by verification of the checksums and the signature. If the AC checks out, the TOC is signed by the receiving AOS and sent back as a receipt which finalizes the migration. Signing is primarily intended for integrity verification and protection against tampering when an AC is shipped over a weakly protected channel. The signed receipt can in addition be stored by AOS as a potential defense against claims that it sent a different AC than it did.

AOS provides a mechanism for starting up an agent in a controlled way. An agent is started up in what is called a *context*. A context is basically a process. If possible, contexts are jailed. Mansion supports only single-process agents, i.e. one agent per context. However, other middleware systems may use a context to run multiple agents in. A context with more than one agent is called an agent server. Jailing (or sandboxing) is used to make sure that a context can only do operations (make RPC calls) to its AppMW via a predefined IPC channel and cannot make connections to external programs.

Depending on the available user-level sandboxes (e.g., a JVM implementation) and the availability of jailing, a given AOS instance supports startup of contexts (agents) written in one or more base languages, such as Java, Python or Safe-Tcl. A configuration file is used to specify the specific interpreters, or virtual machines, and startup options (e.g., for sandbox configuration) that are to be used for specific code subtypes. If no entry for a given code subtype exist, corresponding agent code segments will not be started up.

6. Self-certifying Identifiers

In Mansion, principals are represented by public keys. Principals can be users, agents or processes. A *Self-certifying Identifier (ScID)* [7] is the base32 encoded 160 bit SHA-1 hash of a (PKCS#1/RFC3280 encoded) public RSA key. The base32 encoding yields a 32 byte character string which is filename safe, i.e., the characters used in the base32 encoding contains characters that can be used in a filename or in a URL. ScIDs have a fixed size independent of the public key's length.

The public key corresponding to a ScID is stored in a self-signed certificate. Principals named using ScIDs are:

- *Owners.* An OwnerID is a shorthand for an agent or other entity owner - this is the SHA-1 hash of the public key of the owner. Often, ownerIDs are used in access control policies or lists. An agent owner's certificate and OwnerID are sent along with an agent in its AC.
- *Endpoints.* An EndpointID indicates an individual (nonreplicated) AOS or AppMW process or service, which has its own public / private keypair and a valid self-signed certificate.
- *Zones.* A ZoneID is used to securely name a group of processes which share a single ScID.

A ZoneID corresponds to the SHA-1 hash of the public zone key of the zone administrator. All zone members (i.e., processes in a zone) have their own private/public keypair. This is important, as it makes it possible to distinguish individual zone members. Each zone member has a *zone certificate*, signed using the private zone key, which contains the process'es public key and has an expiration date. The zone certificate proves that the process is part of the zone indicated by ZoneID. In Mansion, ZoneIDs are used to authenticate AppMW (Mansion middleware) level processes and services.

By using a separate key pair for each member, it is possible to identify a malicious zone member and remove it from a zone, or otherwise make changes to zone membership, without changing the ZoneID. The validity of the zone certificate is checked at connection time.

ScIDs can be created and used in a completely decentralized manner. Anyone can create a new zone by creating a public / private key pair and issuing zone certificates without needing any central authority. Note that because ScIDs are self-certifying, there is no need for an external, trusted, binding between a key and a name, such as needed in e.g., x509. An application has complete freedom of creating trust overlay structures using ScIDs.

We wrote a library, zonelib[1], based on the OpenSSL toolkit, for automatically setting up secure (encrypted, reliable), authenticated channels based on Zone or EndpointIDs. Whether the target is a zone member or endpoint is automatically detected. When a connection is set up to a process, it is requested for its endpoint or zone member certificate, and if applicable the zone's public key certificate. The relevant certificate's key is matched against ScID. The server asks for the same information if bidirectional authentication is required. An RSA authentication and key exchange protocol is invoked based on the exchanged public key(s) to set up a secure, authenticated channel.

7. Per World Trust Infrastructure

In Mansion, ScIDs are used to create a world-wide trust infrastructure. The world owner's key is the root of the world's PKI. The world owner signs lists containing the ScIDs of registered zones, WEDs, agent management stations, and other (trusted) principals in a world, and issues them to all members of the world.[2]

Generally, the world owner requires registrants to provide details regarding (world-defined) properties of the zone (or other ScID) they want to register. In that sense, the world administrator acts as a 'traditional' CA, which binds properties to keys (ScIDs). For example, a zone administrator's company name, or a zone's intended purpose may be given in the world's zone list. This is also the way in which zones can be mapped onto existing administratives domains.

1 http://www.cs.vu.nl/~guido/projects/zonelib/

2 Those lists are part of the *World Information System (WIS)*, which essentially consists of the set of datastructures and documents which are needed to make a world work. Its implementation will be briefly touched upon in section 11.

A zone must be on the world's zone list if it is used in a world. Zones that breach confidence can be removed from the world's zone list; only registered zones may be used in a world, for example, in a section's distribution policy.

The fact that ScIDs are registered does not mean that every principal trusts them. For example, a section distribution policy only contains zones which are trusted by the section owner. Similarly, an agent owner can select zones it trusts at world entrance, and place them in a *trusted zone list* embedded in the agent's agent container. If an agent has a trusted zone list, it may only be physically migrated to zones in this list.

A section owner can take action (i.e., remove a zone from the section's distribution policy) if trust is breached, without involving the world owner. Zone members can be removed from a zone if they breach the trust of the zone's administrator, by blacklisting or expiration of their zone certificate[3].

Individual zone members are not known a-priori, they are listed in a (not necessarily trusted) location service (e.g., DNS); the middleware authenticates zone members when a connection is made.

8. Secure Communication

AOS is used to set up protected AOS-to-AOS channels for the purpose of shipping an AC and to facilitate secure AppMW-to-AppMW level communication. Mutually authenticated, encrypted AOS-level channels are set up using an (EndpointID based) authentication and key exchange protocol at the AOS level. Session management and multiplexing connections over a single encrypted AOS-to-AOS connection avoid the expense of key generation for every connection.

An AppMW-level middleware process or service (e.g., naming service) has an *index* relative to AOS which indicates the specific process that the connection is intended for. In case of agent migration, *index* indicates the AppMW for which an AC shipped by AOS is intended. Agents in turn have an index relative to an AppMW level process. (fig. 4).

In Mansion, a separate zone authentication step (sec. 6) is made over the AppMW-to-AppMW channel set up by AOS to authenticate the AppMW level process as a zone member. An authenticated and encrypted channel can also be set up separately at the AppMW level when end-to-end security (e.g., secrecy) is required.

3 The address of a blacklist server may be stored in the zone member's certificate.

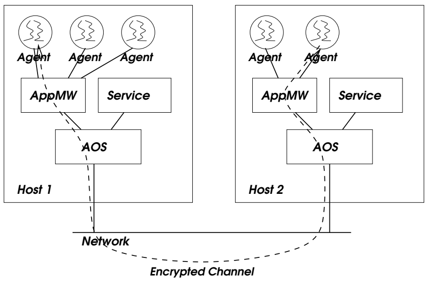

Figure 4. Communication through Middleware.

9. Application Middleware Security

This section explains application middleware level security, with an emphasis on Mansion security mechanisms. Most mechanisms described are also applicable outside the context of Mansion.

9.1. Agent Authentication

Authentication is a problematic issue in mobile agent systems. In general, mobile agents cannot carry private keys or other secrets such as capabilities with them, with which they can authenticate themselves. An agent's code and data, if unencrypted, can be inspected by any host on the agent's itinerary, so secrets can be easily extracted from the agent.

Some solutions exist which may be useful to hide secrets such as private keys in an agent, e.g., code-obfuscation (cloaking) or time-limited blackbox [15] techniques. However, none of these techniques are usable for general-purpose heterogenous agents at this time [13].

Agent authentication in Mansion is based on code signing using the *Agent Passport (AP)* (see also [23]) concept. An AP is composed of a set of signatures of the agent's code[4] and the public key certificate(s) of the signer(s). In particular, the agent's owner signs the Agent Passport; this signature declares that the agent has been sent into the system on behalf of this owner so a middleware that receives the agent can find out which principal owns the agent.

Using the owner's code-signature in the agent passport, the agent's *OwnerID* can be verified. OwnerID is used in access control lists, among other things.

The binding between an agent's AgentID, initial data segments, code segments, AP and other segments such as the agent owner's public key, is created by the trusted world entrance daemon which signed the

4 This can be executable or interpretable code stored inside the AC, or it can be a pointer to a server from which the code can be fetched (e.g., a UR).

agent's AC at world entrance. This binding is important to prevent an attacker from launching a copy of an agent with original code and AP, but otherwise different, to impersonate the agent's owner. A WED authenticates the agent's owner prior to signing the AC to avoid this attack.

Another signer of code segments may be the agent's author, and/or the signature of a code verification company. Using these signatures, an AppMW/AOS combination can establish trust in the safety of an agent's code in absense of trused sandboxes or OS-level jailing. Code signing is particularly imporant for supporting binary code, but may also help protect against exploitation of weaknesses in user-level sandboxes.

9.2. Authorization and Access Control

Objects are protected by distribution and access control policies. Each object has an access control list (ACL) which specifies which principals may access which methods of the object. An ACL contains a list of *OwnerID - access rights* entries. Access rights are expressed in a bitmap, where each bit specifies if access is allowed (1) or not (0). The bits in the bitmap correspond to the methods of the object as defined in the object's class definition. An example is: *3r7xx3q4aynsljossb5kywfcrnvkwwvy: 110011*. Here, agent owner *3r7xx3q4aynsljossb5kywfcrnvkwwvy* has the right to access the first two methods of the object, and the last two. Method 3 and 4 may not be invoked by agents of this owner. ACLs can contain a default entry for unknown agent owners.

Access control is enforced by the Mansion middleware on behalf of the object. Mansion middleware can only mediate access to an object if it is allowed access itself, which depends on a (section wide) distribution policy expressed in terms of zones. An object in Mansion in fact resides in a protected *Object Server* which runs as a separate AppMW level process in the same zone as the Mansion middleware. Details regarding secure binding and replication of objects [3] are hidden inside the object server and are outside the scope of this paper.

Room entrance is controlled by the target room RMO's ACL. If the target room is in the agent's current section, the RMO's ACL is verified by the middleware prior to registering the agent in the room. If the agent's OwnerID (or a default entry) is not present in the ACL, entering the room will fail. If it is present, the agent is unregistered from its current room and registered in the target room. If the target room is not accessible from the agent's current zone, the target room's ACL is checked as part of the agent transfer protocol prior to actual shipment of the agent's AC.

9.3. Confined Rooms

Mansion provides a simple but powerful concept to control information flow called a *confined room.* If an agent enters a confined room, its communication with the outside world is cut off. An agent cannot communicate with any other agent or make any changes (e.g., remove or add data segments) to its AC. It can only export information from the room via a special agent in the confined room called the 'guardian agent.'

A confined agent can interact with objects in its room, inspect data freely, and communicate with other agents in the room. To export information from the confined room, the agent has to tell the guardian agent, for example by giving an index into a list of exportable documents. It can not pass data directly to the guardian to prevent covert channels via information hidden in this data by an agent. Once an agent has exited the confined room (it is restarted), it can ask the guardian agent for copies of the information it requested while it was inside, possibly in return for payment.

As an example, a photo database may contain high-resolution scans of paintings, available to researchers. The database owner may welcome agents to inspect the pictures, but may not want the pictures to be exported to a client's machine for copyright reasons. The database owner can place its pictures inside a (database) object in a confined room, to which researchers can send agents. Those agents can then inspect the database on any property (not just indexed terms), yet exporting pictures from the database is under complete control of the database owner. Agents only have to pay for the pictures they actually require.

10. Agent Protection Mechanisms

By nature, mobile agents are vulnerable [13]. In particular, a host that an agent visits may be malicious in various ways, either passive by trying to extract information from the agent, or more active, by deleting or trying to modify (parts of) an agent, feeding it false information, or sending the agent on a different route than it would do when operating according to its own, internal logic. Another avenue for attack involves attacking the system's agent location service, such that communication to agents may be tampered with or sidetracked to an interposing agent. This section presents some solutions to these problems.

10.1. Agent Management Service

The AMS is a set of services running in a single zone, which keep track of various aspects of an agent during its lifetime in a world. The primary service in an AMS is a location (lookup) service for the contact addresses of the agents managed by the AMS. Other services can include an auditor process (sec. 10.2) or a notary process used in commercial transactions [12]. Each world may have multiple AMSes, each running in its own zone; each AMS can keep track of multiple agents. AMSes can be replicated to increase availability and reliability.

Each agent has an AgentID which consists of the ZoneID of the agent's AMS, and an index relative to the AMS (indicating the agent). Any middleware that needs to contact an agent (to establish inter-agent communication) can request authoritive contact information from that agent's AMS. Middleware can authenticate the AMS using the ZoneID in AgentID.

10.2. Audit Trails

An audit trail is a secure log of the changes that were made to an agent's AC throughout its itinerary. Audit trails are important for mobile agents that migrate over a multihop itinerary of not necessarily trusted middleware processes and hosts. As an example, persistent segments may be used by an agent to store lowest price information on some product. Different companies on the agent's itinerary may have an incentive to remove or change existing persistent segments in the AC to make their own offer look better than earlier ones, even though their offer is really worse. Clearly, it is important to detect tampering of persistent segments in this way. Audit trails are used to detect if and where an AC has been tampered with.

Mansion provides an audit trail mechanism which helps to protect against tampering an agent's persistent state throughouth a multihop itinerary. The audit trail is based on the AC's TOC which describes all segments in the AC (sec. 5).

To establish an audit trail, each time an agent migrates to another middleware, the agent's current middleware signs the agent's TOC (reflecting the AC's current content) with its private key. In addition to the signed TOC, the public key certificate (chain) of the signer is stored in the AC to facilitate identification of the signer. By retaining old TOCs as part of the AC (before the new one is signed), a complete audit trail is established of the changes that were made to the AC on its itinerary. The trusted world entrance daemon signs the first TOC; every itinerary and audit trail starts at a WED (all WEDs are known in a world).

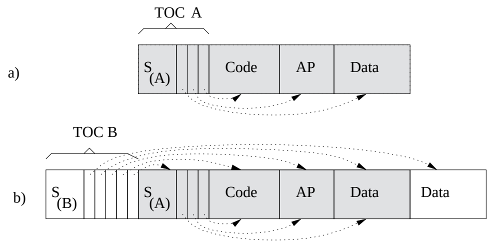

Figure 5. Development of an Audit Trail

An example audit trail is shown in fig. 5. 5a shows the AC as it arrived at middleware B. On middleware B, a data segment was added. TOC A has become part of the AC as a persistent segment of type *toc*. Fig. 5b shows the TOC as it leaves middleware B. A comparison of TOC A and TOC B show exactly what changes were made to the AC on middleware B.

In addition to verifying that no persistent segments were changed or removed, it is possible to store a more general 'AC-change' policy in a persistent segment in the AC and evaluate that with every migration. An AC-change policy may, for example, specify that only a maximum number of transient segments may be removed at each hop. This can be a used to limit the amount of e-cash that can be spent at each hop [12]. Another rule which can be verified using the audit trail is that the agent has only been sent to zones in the agent's trusted zone list (sec. 7).

Some attacks are possible on the audit trail if it is stored only in the AC, particularly rollback to an earlier state of the AC when there are cycles in the agent's itinerary. Different methods exist to prevent these attacks, such as sending the signed TOCs to an external auditor process in addition to storing them in the AC. Other solutions were discussed in [12] and [19].

10.3. The Agent Handoff Protocol

To protect the agent as well as its contact information in the AMS, an *agent handoff protocol* is used as part of Mansion's agent migration protocol.

The handoff protocol is based on AC integrity verification using the audit trail / TOC verification mechanism discussed in section 10.2. If an AC appears tampered with, either because its signature or TOC is not correct or because the audit trail shows that illegitimate changes were made to the AC, the migration protocol will be aborted. An agent's AgentID and contact information are initially registered in the AMS by the agent's world entrance daemon. On migration, the

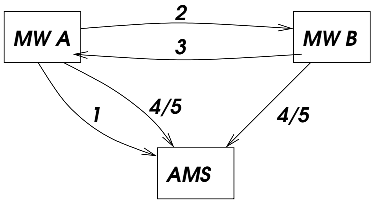

Figure 6. The Handoff Protocol

sending and the receiving middleware have to exchange proof that they agree to the agent's migration before the agent's contact information in the AMS is updated.

The handoff protocol protects both the agent and its contact information in the AMS. Agent migration of an agent is only official when the agent's location is updated at its AMS. Not accepting a tampered-with AC effectively confines an agent to its current middleware; this stops a maliciously altered agent (AC) from migrating any further in an incorrect state.

The handoff protocol works as follows (fig. 6). To initiate migration, middleware A sends an *init_migration(AgentID, target)* message to the agent's AMS (step *1*). AgentID's AMS knows the agent's current middleware's location and ScID (ZoneID). It uses this information to authenticate the init_migration request; only a request coming from the agent's current middleware is accepted. *Target* contains the target middleware's address and ScID. Next the agent's AC is sent to B *(2)*[5]. B verifies that the AC's TOC corresponds to the segments in the AC and that no illegitimate changes were made to the AC (e.g., no persistent segments were removed or changed) in the way described in section 10.2

If B accepts the agent, it also signs the TOC and sends the (now doubly signed) TOC to middleware A as a receipt *(3)*. To finalize the agent contact information update, A and B both send an authenticated (signed) *commit_migration* message to the agent's AMS *(4)*. If A or B do not agree to the migration (for whatever reason), they can abort the migration transaction at any time in the protocol *(5)*.

In addition, A sends the doubly signed TOC to an external auditor process, which is part of the agent's AMS and runs in the AMS'es zone. The auditor process timestamps and archives the received TOCs as a secure record of an agent's whereabouts over time.

5 For simplicity we assume that the AC is directly sent to B. In reality, the AC is sent to the AOS kernel used by B and handed off to MMW. Details of how this detail is solved are outside the scope of this paper.

11. Decentralization, Scalability and Security

The scalability of a world depends on the way in which information is distributed. Many parts of a world can be managed decentralized. Examples of decentrally managed world parts are sections, which in turn make use of decentrally administered zones. Furthermore, most information is only stored in untrusted services, for which existing large-scale infrastructures such as DNS or the web may be used. This is an advantage of using connection-time authentication using ScIDs, as explained in section 6.

A few documents and data structures are managed globally, which bind the world together. These documents and data structures are conceptually part of a global *World Information System (WIS)*, which is the only world part which has to be managed directly by the world designer. Example data in the WIS are the world's zone list, zone revocation list, AMS list, and WED list.

The WIS implementation details are purposefully left open to allow for different implementations. For example, when a WIS document is updated or has expired, it can be uploaded to section entry daemons or some zone member processes. These processes can distribute the information as soon as they are contacted by other middleware processes, so that the information perculates over the system eventually. WIS information is mostly CA related information and generally requires no absolute consistency, as the information itself has an expiration date which can be tuned to the application's consistency or security requirements.

When WIS information has expired, it should no longer be used, and action should be taken to obtain up-to-date information. A small set of trusted servers (e.g., in a trusted zone owned by the world designer) per world may suffice to answer requests for immediate provision of such information.

12. Privacy and Anonymity

Mansion protects against tracing of the agent's whereabouts and interests. In a room an agent is not known by its global AgentID, but by its *EntityID* (sec. 3.1) EntityID is used as a pseudonym to connect to an agent, so that agents do not need to advertise their AgentIDs in every room they enter. Agents can pass their AgentID to other agents once a 'local' contact has been established.

An agent's logical itinerary (accessed rooms) is not registered in any global Mansion service or in the agent's AC. Only the agent's own (current) middleware knows what the agent has done and where it's been for the duration that the agent was there. An agent's physical address can be looked up in the agent location service. However, as an agent does not always need to migrate physically when it follows a hyperlink (many hyperlinks will be relative to the agent's current section), not much information is obtained from polling an agent's contact information from its AMS in general.

Anonymity is not currently provided by Mansion, although mechanisms based on world entrance daemons which act as an anonymizer are conceivable. For example, the agent's code signature in the AP could be replaced by a signature using a WED's key or a temporary key. Such a key cannot be traced back to the agent's owner, except possibly through the WED. If the rule that an agent owner needs to authenticate itself to the WED is loosened, agents can even be shipped to a WED through various additional layers of anonymization, similar to anonymous remailer systems. Note that the latter mechanism makes it impossible to enforce world access restrictions based on (preregistered) agent owner authentication, such as a limit on the number of agents allowed into a world per agent owner.

13. Related Work

Self-certifying identifiers were first introduced in the secure file system (SFS) [7]. In SFS, ScIDs are used as part of self-certifying pathnames. However, SFS is based on authenticating single hosts on which files are placed, rather than on authenticating groups of processes (zones) in a location-indpendent way, as we do.

Telescript [17] was the first commercial mobile agent system, and pioneered many of the concepts common to all mobile agent systems, including Mansion. Telescript had places, comparable to rooms, and regions, comparable to sections/zones. Telescript had no hyperlinks (nor does any other agent system). Places in Telescript mapped onto single machines. Telescript featured an interpreted, object-oriented language which contained primitives to migrate to a place (using strong migration), and to communicate or colocate (meet) with other agents. Most if not all mobile agent systems have similar concepts as Telescript.

With the advent of Java at the end of the '90s, a large number of Java-based mobile agent systems were designed and built [19, 20, 8, 16]. Those systems are largely dependent on the security and platform neutrality provided by Java. Most current systems use strong migration based on a modified JVM. JavaSeal [4] provides a 'kernel' providing general abstractions usable for programming secure Java agent systems. Ajanta [19] is a Java-based agent system which has a similar audit trail mechanism as Mansion. However, the Ajanta scheme is limited by the fact that at each hop the TOC is encrypted. Therefore, audit trail inspection can only take place when an agent gets home.

Most systems focus primarily on the (mobile) agent paradigm, although some systems also support distributed objects [11]. Few if any existing system separates logical from physical location (allowing for replication or remote access to logical locations) in a way similar to Mansion.

Only few systems support heterogenous agents, an important example being D'Agents (formerly Agent-Tcl) [24], which supports Tcl, Scheme and Java agents. An earlier system that supported heterogenous agents was TACOMA [6]. TACOMA used a 'folder' concept, similar to an AC (but unprotected), which could be used to contain several implementations of an agent and its associated data. Ara [23] supports heterogenous agents and strong migration (using modified interpreters), distinguishes mutable and immutable parts of an agent's execution state, and has an agent passport concept similar to Mansion.

Most systems emphasize an 'open' model, with loosely coupled places (e.g., [22]). In open models, an agent's environment is dynamic and offers little structure to agents. Mansion, in contrast, bases much of its security (notably its trust infrastructure) on a closed world concept, which is also used to structure applications. Hyperlinks and rooms provide structure to application developers and agents roaming the world.

14. Summary

In this paper, a security architecture for the Mansion mobile agent framework was presented. Mansion provides a clear logical paradigm for structuring worlds consisting of sections, rooms and hyperlinks. Confined rooms show how Mansion's logical structure can be used to control information flow.

A layered middleware design is introduced for supporting heterogenous agents and platforms. The lowest layer (AOS) hides the underlying platforms from higher layers and provides minimal functionality common to most mobile agent systems: secure agent storage, agent transport, secure connections and secure process execution. On this layer we build various Mansion (application) specific security mechanisms: authentication and authorization, control over migration, auditing of changes to agents, and protection of the agent's contact information.

Self-certifying identifiers are used pervasively throughout the system to simplify key management, authentication, and the specification of access control lists and distribution policies. Mansion worlds are mapped onto an overlay network consisting of securely named groups of processes called zones. Zone properties are defined in an application-dependent manner by the world designer. Despite the decentralization inherent in parts of the design, worlds are closed: principals in a world are recognized and trusted only if they are part of the world's trust infrastructure, which is rooted by the world owner's public key. A world owner can delegate most functionality and administrative tasks (e.g., AMS management) to any subsidiary by including it in a world membership list signed by the world owner.

Acknowledgements

We like to thank Benno Overeinder, Etienne Posthumus, and David Mobach for useful discussions on the model. Etienne Posthumus implemented AOS. Adam Balogh implemented Linux jailing. Bruno Crispo, Melanie Rieback, and Ruediger Weis gave useful feedback on early drafts of this paper. We also thank the anonymous reviewers for their helpful comments, and Stichting NLnet for their support.

References

[1] A. Bakker; M. van Steen; and A.S. Tanenbaum. From Remote Objects to Physically Distributed Objects. *Proc. 7th IEEE Workshop on Future Trends of Distributed Computing Systems*, December 1999. pp. 47-52.

[2] A. Balogh. New Object Server for Globe. *Master's Thesis, Vrije Universiteit, The Netherlands*, 2003.

[3] B.C. Popescu; M. van Steen; A.S. Tanenbaum. A Security Architecture for Object-Based Distributed Systems. *Proc. 18th IEEE Annual Computer Security Applications Conference*, December 2002. pp. 161-171.

[4] C. Bryce and J. Vitek. The JavaSeal Mobile Agent Kernel. *Autonomous Agents and Multi-Agent Systems 4*, 2001. pp. 359-384.

[5] D. Chess; B. Grosof; C. Harrison; D. Levine; C. Parris; G. Tsudik. Itinerant Agents for Mobile Computing. *IEEE Personal Communications*, 4(5):34–49, October 1995.

[6] D. Johansen; R. van Renesse; F.B. Schneider. Operating systems support for mobile agents. *5th Workshop on Hot Topics in Operating Systems*, 1995. pp. 42-45.

[7] D. Mazieres; M. Kaminsky; M.F. Kaashoek; E. Witchel. Separating Key Management From File System Security. *17th ACM Symposium on Operating Systems Principles*, 1999. pp. 124-139.

[8] D. Milojicic; F. Douglis; R. Wheeler, eds. Mobility: processes, computers and agents. *ACM Press*, 1999.

[9] D.S. Wallach; D. Balfanz; D. Dean; E.W. Felten. Extensible Security Architectures for Java. *16th ACM Symposium on Operating Systems Principles*, 1997. pp. 116-128.

[10] G. Back and W. Hsieh. Drawing the Red Line in Java. *Workshop on Hot Topics in Operating Systems (HotOS VII)*, 1999. pp. 116-121.

[11] G. Glass. ObjectSpace Voyager Core Package Technical Overview. *www.objectspace.com.* In Milojicic et al., (eds). Mobility: processes, computers and agents, ACM press, 1999, pp. 611-627.

[12] G. van 't Noordende; F.M.T. Brazier; A.S. Tanenbaum. A Security Framework for a Mobile Agent System. *2nd Int'l Workshop on Security of Mobile Multiagent Systems (SEMAS)*, July 2002. Bologna, Italy. pp. 43-50.

[13] G. Vigna (ed.). Mobile Agents and Security. *LNCS 1419*, 1998. Springer-Verlag.

[14] L. Gong. *Inside Java 2 Platform Security: Architecture, API Design, and Implementation.* Addison-Wesley, June 1999.

[15] F. Hohl. Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts. *Mobile Agents and Security*, 1998. LNCS 1419, Springer-Verlag pp. 154-187.

[16] J. Baumann; F. Hohl; M. Strasser; K. Rothermel. Mole - Concepts of a Mobile Agent System. *Technical Report, Universität Stuttgart*, August 1997.

[17] J.E. White. Telescript Technology: Mobile Agents. *White paper, General Magic*, 1996.

[18] J.K. Ousterhout; H.Y. Levy; B.B. Welch. The Safe-Tcl Security Model. *Mobile Agents and Security*, 1998. LNCS 1419, Springer-Verlag.

[19] N. Karnik and A. Tripathi. Security in the Ajanta Mobile Agent System. *Software - Practice and Experience 31(4)*, 2001. pp. 301-329.

[20] D. Lange and M. Othima. Mobile Agents with Java: The Aglet API. *World Wide Web 1(3)*, September 1998.

[21] N. Provos. Improving Host Security with System Call Policies. *Proc. 12th USENIX Security Symposium*, August 2003. pp. 257-272.

[22] N.J.E. Wijngaards; B.J. Overeinder; M. van Steen; F.M.T. Brazier. Supporting Internet-Scale Multi-Agent Systems. *Data and Knowledge Engineering 41(2-3)*, 2002. pp. 229-245.

[23] H. Peine. Security Concepts and Implementation for the Ara Mobile Agent System. *7th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises*, June 1998.

[24] R.S. Gray; D. Kotz; G. Cybenko; D. Rus. D'Agents: Security in a Multiple-language, Mobile-agent System. *Mobile Agents and Security*, 1998. LNCS 1419, Springer-Verlag pp. 154-187.

[25] W. Binder and V. Roth. Secure mobile agent systems using Java: where are we heading? *Proceedings of the 2002 ACM Symposium on Applied Computing*, 2002. pp. 115-119.