Security for Mobile Agents: Authentication and State Appraisal [*]

William M. Farmer, Joshua D. Guttman, and Vipin Swarup

The MITRE Corporation
202 Burlington Road
Bedford, MA 01730-1420

{farmer, guttman, swarup}@mitre.org

Abstract. Mobile agents are processes which can autonomously migrate to new hosts. Despite its many practical benefits, mobile agent technology results in significant new security threats from malicious agents and hosts. The primary added complication is that, as an agent traverses multiple hosts that are trusted to different degrees, its state can change in ways that adversely impact its functionality. In this paper, we discuss achievable security goals for mobile agents, and we propose an architecture to achieve these goals. The architecture models the trust relations between the principals of mobile agent systems. A unique aspect of the architecture is a "state appraisal" mechanism that protects users and hosts from attacks via state modifications and that provides users with flexible control over the authority of their agents.

1 Introduction

Currently, distributed systems employ models in which processes are statically attached to hosts and communicate by asynchronous messages or synchronous remote procedure calls. Mobile agent technology extends this model by including mobile processes, i.e., processes which can autonomously migrate to new hosts. Numerous benefits are expected; they include dynamic customization both at servers and at clients, as well as robust remote interaction over unreliable networks and intermittent connections [5,11,17].

Despite its many practical benefits, mobile agent technology results in significant new security threats from malicious agents and hosts. In fact, security issues are recognized as critical to the acceptability of distributed systems based on mobile agents. The primary added complication is that, as an agent traverses multiple machines that are trusted to different degrees, its state can change in ways that adversely impact its functionality. In this paper, we will discuss achievable security goals for mobile agents, and we will propose an architecture to achieve these goals. We use existing theory—the distributed authentication

[*] Work supported by the MITRE-Sponsored Research Program.
Appeared in the Proceedings of the European Symposium on Research in Computer Security (ESORICS), pages 118–130, September 1996.

theory of Lampson et al. [10]—to clarify the architecture and to show that it meets its objectives.

The process of deducing which principal made a request is called *authentication.* In a distributed system, authentication is complicated by the fact that a request may originate on a distant host and may traverse multiple machines and network channels that are secured in different ways and are not equally trusted [10]. The process of deciding whether or not to grant a request—once its principal has been authenticated—is called *authorization.* The authentication mechanism underlies the authorization mechanism in the sense that authorization can only perform its function based on the information provided by authentication, while conversely authentication requires no information from the authorization mechanism.

State Appraisal The unique aspect of our architecture is a "state appraisal" mechanism. State appraisal lies above the lower authentication layer, and provides input to an authorization mechanism like those of static distributed systems. When an agent arrives at a new site of execution, it must decide what privileges it will need at that site. A state appraisal function computes a set of privileges to request as a function of the state of the agent when it arrives at the new execution site. After state appraisal has determined which permissions to request, the authorization mechanism on the new execution site may then determine which of the requested permissions it is willing to grant.

The state appraisal mechanism serves several purposes:

1. It can protect a host from attacks, when these attacks alter the states of agents in dangerous but detectable ways;
2. It can protect an agent's author and sender (the user who dispatched this agent) from misuse of the agent in their name via dangerous but detectable state modification;
3. It can check whether an agent's state meets important state invariants;
4. It enables an agent's privilege at a new host to be dependent on the agent's current state, and therefore on the task to be carried at that host.

Not all state alteration can be detected, because agents travel to new hosts to acquire information that is not available elsewhere. Some deceptive alterations of the agent's state will be indistinguishable from the normal result of different (but possible) information on the remote host. However, alterations that would cause the agent to become harmful—either to sites it visits later or else to the user who dispatched it—can frequently be detected.

Our work contrasts with other work on mobile agent security [12–14,16,17] because it focuses on the state information that agents carry with them. We also explain the relationship between state appraisal, authentication, and authorization. We emphasize agents written by known software developers, which we think will be the predominant way of using software agents; most other work looks at security for agents written by unknown parties.

We avoid making assumptions about the environment offered by different sites of execution. In particular, we do not assume that the agents can count on protection from the operating system at those sites. We believe that many of the most useful applications for mobile agents will be in heterogeneous environments. Different sites will use different operating systems, none of them likely to provide a very high level of protection. Moreover, in these heterogeneous environments, some sites will be in hostile or at least competitive relations with each other. Thus, they cannot be expected to leave their operating systems unmodified, if modifying the systems—for instance to defeat security mechanisms—would further their competitive goals.

Mobile Agents and Languages A mobile agent is a program that can migrate from one networked computer to another while executing. This contrasts with the client/server model where non-executable messages traverse the network, but the executable code remains permanently on the computer on which it was installed. Mobile agents have numerous potential benefits. For instance, in a specialized search of a large free-text database, it may be more efficient to move the program to the database server rather than to move large amounts of data back to the client's host.

In recent years, several programming languages for mobile agents have been designed. These languages make different design choices as to which components of a program's state can migrate from machine to machine. In Java [11], only program code can migrate; no state is carried with the programs. In Obliq [1], first-class function values (closures) can migrate; closures consist of program code together with an environment that binds variables to values or memory locations [16]. In Kali Scheme [2], again, closures can migrate; however, since continuations [8,6] are first-class values, Kali Scheme permits entire processes to migrate autonomously to new hosts. In Telescript [17], functions are not first-class values; however, Telescript provides special operations that permit processes to migrate autonomously.

In this paper, we adopt a fairly simple but general model of mobile agents. *Agent interpreters* run on individual networked computers and communicate among themselves using host-to-host communication services. An *agent* consists of code together with execution state. The state may include a program counter, registers, environment, recursion stack, and store; the agent executes by being interpreted by an agent interpreter.

Agents communicate among themselves by message passing. In addition, agents can invoke a special asynchronous "remote apply" operation that applies a closure to arguments on a specified remote interpreter. Remote procedure calls can be implemented with this primitive operation and message passing. Agent migration and cloning can also be implemented with this primitive operation, using first-class continuation values.

Example: Travel Agents, Mobile Agents to Plan Travel We turn now to an example mobile agent system. We believe that it is typical of many—though not of all—ways that mobile agents will be used. We will use the example to extract the most important principles that a security architecture for mobile agents should obey.

Consider a mobile agent that visits sites run by airlines, hotel chains, and rental car companies searching for a travel plan that meets a customer's requirements. We focus on four kinds of hosts: the personal communication systems of the end customers, the hosts run by travel agencies in the traditional sense of organizations that broker travel arrangements, and the hosts run by the competing airlines, hotel chains, and car rental firms. Let us refer to the airlines as Airline 1 and Airline 2, which we assume for the sake of this example do not share a common reservation system.

The mobile agent is programmed by the travel agency. A customer dispatches the agent to the Airline 1 server where the agent queries the flight database. With the results stored in its environment, the agent then migrates to the Airline 2 server where again it queries the flight database. From either of these hosts, it may visit a hotel host, where it may be eligible for a special deal depending on whether the customer will be traveling on an allied air carrier. The agent compares flight and fare information, decides on a travel plan, migrates to the appropriate airline and hotel hosts, and reserves the desired flights and rooms. Finally, the agent returns to the customer with the results.

The customer can expect that the individual airlines and hotels will provide true information on flight schedules and fares in an attempt to win her business, just as we assume nowadays that the reservation information the airlines provide over the telephone is accurate, although it is not always complete.

However, the airline servers are in a competitive relation with each other. The airline servers illustrate a crucial principle: *For many of the most natural and important applications of mobile agents, we cannot expect all the participants to trust one another.*

There are a number of attacks they may attempt. For instance, the second airline server may be able to corrupt the flight schedule information of the first airline, as stored in the environment of the agent. It could surreptitiously raise its competitor's fares, or it could advance the agent's program counter into the preferred branch of conditional code. Thus, the mobile agent cannot decide its flight plan on an airline host since the host has the ability to manipulate the decision. Instead, the agent would have to migrate to a neutral host such as the customer's host or a travel agency host, make its flight plan decision on that host, and then migrate to the selected airline to complete the transaction. This attack illustrates a principle: *An agent's critical decisions should be made on neutral hosts, which is to say hosts trusted by its sender.*

A second kind of attack is also possible: the first airline may hoodwink the second airline, for instance when the second airline has a cheaper fare available. The first airline's server surreptitiously increases the number of reservations to be requested, say from two seats to 100. The agent will then proceed to reserve 100 seats at the second airline's cheap fare. Later, legitimate customers will have to book their tickets on the first airline, as the second believes that its flight is full.

This attack suggests a third principle: *a migrating agent can become malicious by virtue of its state being corrupted.*

Moreover, it may not be sufficient to seal certain state components cryptographically, such as the number of seats to be requested, which in our example was two. Rather, we would like to be able to ensure that a state invariant is preserved: in this case, that the sum of the number of seats already booked and those still to be requested should remain constantly the value two. Otherwise, the first airline may be able to send the same agent to the second airline repeatedly, booking two more unnecessary seats on each visit.

On the other hand, some organizations may trust each other, as for instance in the case where they are divisions of the same corporation. An agent arriving from a trusted partner may be subjected to less security examination (to reduce overhead), or it may be accorded special privileges. *Authentication and authorization may be handled specially between hosts with reciprocal trust.*

2 Security Goals

There are some basic constraints on what security goals are achievable. We assume that different parties will have different degrees of trust for each other, and in fact some parties may be in a competitive or even hostile relation to one another. As a consequence, we may infer that one party cannot be certain that another party is running an untampered interpreter. An agent that reaches that party may not be allowed to run correctly, or it may be discarded. The interpreter may forge messages purporting to be from the agent. Moreover, the interpreter may inspect the state of the agent to ferret out its secrets. For this reason, we assume that agents do not carry keys. In particular, we claim that agents *cannot* carry keys in a form that can be used on untrusted interpreters.

Existing approaches for distributed security [9] do allow us to achieve several basic goals. These include authenticating an agent's author and its sender, checking the integrity of its code, and offering it privacy during transmission, at least between interpreters willing to engage in symmetric encryption. We have discussed these points in more detail in [4].

However, at least three crucial security goals remain:

1. Certification that an interpreter has the authority to execute an agent on behalf of its sender;
2. Flexible selection of privileges, so that an agent arriving at an interpreter may be given the privileges necessary to carry out the task for which it has come to the interpreter; and
3. State appraisal, to ensure that an agent has not become malicious as a consequence of alterations to its state.

3 Authentication

Authentication is the process of deducing which principal has made a specific request. In a distributed system, authentication is complicated by the fact that a request may originate on a distant host and may traverse multiple machines and network channels that are secured in different ways and are not equally trusted. For this reason, Lampson and his colleagues [10] developed a logic of authentication that can be used to derive one or more principals who are responsible for a request.

Elements of a Theory of Authentication The theory—which is too rich to summarize here—involves three primary ingredients. The first is the notion of *principal.* Atomic principals include persons, machines, and roles; groups of principals may also be introduced as principals; and in addition principals may also be constructed from simpler principals by operators. The resulting compound principals have distinctive trust relations with their component principals. Second, principals make *statements*, which include assertions, requests, and performatives.[1] Third, principals may stand in the *"speaks for"* relation; one principal P_1 speaks for a second principal P_2 if, when P_1 **says** s, it follows that P_2 **says** s. This does not mean that P_1 is prevented from uttering phrases not already uttered by P_2; on the contrary, it means that if P_1 makes a statement, P_2 will be committed to it also. For instance, granting a power of attorney creates this sort of relation (usually for a clearly delimited class of statements) in current legal practice. When P_1 speaks for P_2, we write $P_1 \Rightarrow P_2$. One of the axioms of the theory allows one principal to pass the authority to speak for him to a second principal, simply by saying that it is so:

$$(P_1 \textbf{ says } P_2 \Rightarrow P_1) \supset P_2 \Rightarrow P_1$$

This is called the *hand-off* axiom; it says that a principal can hand his authority off to a second principal. It requires a high degree of trust.

Three operators will be needed for building compound principals, namely the **as**, **for**, and quoting operators. If P_1 and P_2 are principals, then P_1 **as** P_2 is a compound principal whose authority is more limited than that of P_1. P_2 is in effect a *role* that P_1 adopts. In our case, the programs (or rather, their names or digests) will be regarded as roles. Quoting, written $P \mid Q$ is defined straightforwardly: $(P \mid Q)$ **says** s abbreviates P **says** Q **says** s.

The **for** operator expresses *delegation.* P_1 **for** P_2 expresses that P_1 is acting on behalf of P_2. In this case P_2 must delegate some authority to P_1; however, P_1 may also draw on his own authority. For instance, to take a traditional example, if a database management system makes a request on behalf of some user, the request may be granted based on two ingredients, namely the user's identity supplemented by the knowledge that the database system is enforcing some

[1] A statement is a *performative* if the speaker performs an action by means of uttering it, at least in the right circumstances. The words "I do" in the marriage ceremony are a familiar example of a performative. Similarly, "I hereby authorize my attorneys, Dewey, Cheatham and Howe, jointly or severally, to execute bills of sale on my behalf." Semantically it is important that requests and performatives should have truth values, although it is not particularly important how those truth values are assigned.

constraints on the request. Because P_1 is combining his authority with P_2's, to authenticate a statement as coming from P_1 **for** P_2, we need evidence that P_1 has consented to this arrangement, as well as P_2.

Mobile agents require no additions to the theory presented in [10]; the theory as it exists is an adequate tool for characterizing the different sorts of trust relations that mobile agents may require.

Atomic Principals for Mobile Agents Five categories of basic principals are relevant:

- The *authors* (whether people or organizations) that write programs to execute as agents;
- The *programs* they create, which, together with supplemental information, are signed by the author;
- The *senders* (whether people or other entities) that send agents to act on their behalf. A sender may need a trusted device to sign and transmit agents;
- The *agents* themselves, consisting of a program together with data added by the sender on whose behalf it executes, signed by the sender;
- The *interpreters* that execute agents; they may transfer the agents among themselves, and may eventually return results to the sender.

Naturally, an implementation will also require one or more certification authorities, but their role is perfectly standard and we will not discuss them further.

The Natural History of an Agent There are three crucial types of events in the life history of an agent. They are the creation of the underlying program; the creation of the agent; and migration of the agent from one execution site to another. These events introduce compound principals built from the atomic principals just given.

Program Creation. The author of a program prepares source code and a state appraisal function `max` for the program. The function `max` will calculate the maximum safe permissions to be accorded an agent running the program, as a function of its current state.

In addition, an access control list (or other mechanism) may also be included for determining which users are permitted to send the resulting agent.

After compiling the source code for the program and its state appraisal function, the author C then compiles these pieces, constructs a message digest D for the result, and signs that with her private key. By doing so, she permits the program to make statements on her behalf, in her role as author of this program. That is, D, when attributing a statement to C, may speak for the compound principal C **as** D (formally, $D \,|\, C \Rightarrow C$ **as** D).[2]

[2] Since we regard D as a *name* of the program of which it is a digest, we regard actions performed by the program while executing as utterances of D.

Agent Creation. To prepare a program for sending, the sender attaches a second state appraisal function, called the request function; this will calculate the permissions the sender wants an agent running the program to have, as a function of its current state. We will call the sender's state appraisal function `req`. For some states Σ, `req`(Σ) may be a proper subset of `max`(Σ); for instance, the sender may not be certain how D will behave, and she may want to ensure she is not liable for some actions.

The sender may also include an interpreter access control list (or other mechanism) specifying interpreters that are allowed to run the resulting agent on the sender's behalf, either via delegation or hand-off.

The sender S attaches her name, and then she computes a message digest A for the following items: the program, its digest D, the function `req`, S's name, and a counter S increments for each agent she sends. She signs the message digest A with her private key.

This signature says that the resulting agent A may make statements for the compound principal A **for** S. This is a delegation, which requires some sort of acceptance by A. This acceptance amounts to permission for S to use A. The acceptance may then be attested by the access control list, mentioned above as a component of D, for determining the users authorized to send agents built from D. Alternatively, if C wants D to be usable by the general public, this acceptance may be vacuous. As a consequence of the delegation, A, when attributing a statement to S, may speak for the compound principal A **for** S (formally, $A \,|\, S \Rightarrow A \textbf{ for } S$).

In addition, A can speak for D: Statements made by an agent A are statements made by its program, and D is the digest of that program.

Before the sender dispatches A, she also attaches a list of parameters, which are in effect the initial state Σ_0 for the agent. The state is not included under any cryptographic seal, because it must change as the agent carries out its computation. However, S's request function may impose invariants on the state.

Agent Migration. When an agent is ready to migrate from one interpreter to the next, the current interpreter must construct a message containing the agent A, its current state Σ, the current interpreter I_1, the principal P_1 on behalf of whom I_1 is executing the agent, and a description of the principal P_2 on behalf of whom the next interpreter I_2 should execute the agent starting from Σ.

The authentication machinery can be construed as providing a proof that $I_2 \,|\, P_2 \Rightarrow P_2$. Depending on whether I_2 is trusted by I_1 or by the agent A, four different values of P_2 are possible, expressing different trust relationships.

1. I_1 can hand the agent off to I_2; I_2 will then execute the agent on behalf of P_1. In this case, $I_1 \,|\, P_1 \textbf{ says } I_2 \,|\, P_1 \Rightarrow P_1$. P_2 is P_1.
2. I_1 can delegate the agent to I_2. I_2 will combine its authority with that of P_1 while executing the agent. P_2 is $I_2 \textbf{ for } P_1$.
3. The agent can directly hand itself off to I_2. I_2 will execute A on behalf of the agent. In this case, $A \,|\, S \textbf{ says } I_2 \,|\, (A \textbf{ for } S) \Rightarrow (A \textbf{ for } S)$. This statement may be contained in an interpreter access control list the sender has attached to the agent. P_2 is A **for** S.

4. The agent can delegate itself to I_2. I_2 will then combine its authority with that of the agent while executing A. The delegation statement may be contained in an interpreter access control list the sender has attached to the agent. In this case, P_2 is I_2 **for** A **for** S.

In the first case, I_2 does not appear in the resulting compound principal. This requires I_1 to trust I_2 not to do anything I_1 would not be willing to do. In the third and fourth cases, because the agent itself is explicitly expressing trust in I_2, the resulting compound principal does not involve I_1. The agent trusts I_2 to appraise the state before execution. Assuming that the result of the appraisal is accepted, I_1 has discharged its responsibility.

In all four cases, if P_1 is of the form Q **for** $\ldots(A$ **for** $S)$, where the initial chain of delegations Q **for** $\ldots$ may be vacuous, then P_2 is also of the same form. Since agent creation—which serves as a base case—yields the principal A **for** S, by induction the relevant principal is always of the form shown.[3]

What happens when an interpreter I_2 receives a message requesting that it execute an agent? The message contains an agent (A **for** S), together with its current state Σ and the name of a principal P_2 on behalf of whom I_2 will execute it. I_2 must evaluate the certificates contained in the message to authenticate P_2. This may require it to fetch certificates from a directory service for principals it does not recognize. It will also check the author's signature on the program and the sender's signature on the agent. However, the mechanisms needed for this purpose are well-understood [10]; indeed, they provide a proof that $I_2 \,|\, P_2 \Rightarrow P_2$. We have now met the first of the security goals proposed in Section 2, namely certification that an interpreter has the authority to execute an agent, ultimately on behalf of its sender.

Trust Relationships for Travel Agents We now return to our travel agents example (Section 1) and describe how the various trust relationships of that example can be expressed in our security architecture.

In the example, a travel agency creates a travel reservation program and a state appraisal function while a customer adds a permit request function. The customer also provides certificates to hand off or delegate authority to interpreters. For instance, the customer's certificates may hand off authority to her own interpreter and to a neutral travel agency interpreter (since she trusts them), but her certificates may only delegate authority to Airline 1 and Airline 2 (since they have vested interests). The customer sends the agent to her interpreter, with an initial state containing her desired travel plans.

As its first task, the agent migrates to the Airline 1 host I_1. The agent contains a certificate that directly delegates to Airline 1 the authority to speak on its behalf. Airline 1 accepts this delegation and runs the agent as I_1 **for** A **for** S. In our architecture, this kind of trust relationship is an example of the *agent delegating its authority* to Airline 1 (case 4). This ensures that Airline 1 takes

[3] Indeed, we may regard agent creation as an instance of case 3, where the new interpreter I_2 is the sender's local host.

responsibility while speaking for the agent, for instance, while making airline reservations on the agent's behalf.

The agent now seeks to gather hotel reservation information. Airline 1 owns a hotel chain and has strong trust in its hotels such as Hotel 1. It sends the agent to the Hotel 1 host I_2 and gives Hotel 1 whatever authority it has over the agent. Hotel 1 runs the agent as I_1 **for** A **for** S, which is the principal that I_1 hands it. This kind of trust relationship is an example of Airline 1's *interpreter handing off its authority* to Hotel 1 (case 1). As a consequence of this trust, I_2 may grant the agent access to a database of preferred room rates.

Next, the agent migrates to Airline 1's preferred car rental agency Car Rental 1, whose interpreter is I_3. Since Airline 1 does not own Car Rental 1, it delegates its authority to Car Rental 1 which runs the agent as I_3 **for** I_1 **for** A **for** S. This causes Car Rental 1 to take responsibility while speaking on Airline 1's behalf. It also gives the agent combined authority from I_1 and I_3; for instance, the agent can obtain access to rental rates negotiated for travelers on Airline 1. Airline 1's interpreter *has delegated its authority* to Car Rental 1 (case 2).

The agent now migrates to the Airline 2 host I_4. The agent contains a certificate that directly delegates to Airline 2 the authority to speak on its behalf. Airline 2 accepts this delegation and runs the agent as I_4 **for** A **for** S, again case 4. Airline 1's interpreter I_1 has now discharged its responsibility; it is no longer an ingredient in the compound principal.

Once the agent has collected all the information it needs, it migrates to the customer's trusted travel agency (Travel Agency 1) host I_5 to compare information and decide on an itinerary. The agent contains a certificate that directly hands Travel Agency 1 the authority to speak on its behalf. Travel Agency 1 can thus run the agent as A **for** S. This permits Travel Agency 1 to make critical decisions for the agent, for instance, to make reservations or purchase a ticket. This kind of trust relationship is an example of the *agent handing off its authority* to Travel Agency 1 (case 3).

4 Authorization

The result of the *authentication* layer is a principal P_2 on behalf of whom I_2 has been asked to execute the agent. The purpose of the *authorization* layer is to determine what level of privilege to provide to the agent for its work. The authorization layer has two ingredients. First, the agent's state appraisal functions are executed; their result is to determine what privileges ("permits") the agent would like to *request* given its current state. Second, the interpreter has access control lists associated with these permits; the access control lists determine which of the requested permits it is willing to *grant*.

We will assume that the request is for a set α of permits; thus, a request is a statement of the form *please grant* α. In our approach, agents are programmed to make this request when they arrive at a site of execution; the permits are then treated as capabilities during execution: no further checking is required.

We distinguish one special permit `run`. By convention, an interpreter will run an agent only if it grants the permit `run` as a member of α.

The request is made by means of the two state appraisal functions. The author-supplied function `max` is applied to Σ returning a maximum safe set of permits. The sender-supplied appraisal function `req` specifies a desired set of permits; this may be a proper subset of the maximum judged safe by the author. However, it should not contain any other, unsafe permits. Thus, we consider P_2 to be making the conditional statement:

$$\textit{if}\ \texttt{req}(\Sigma) \subseteq \texttt{max}(\Sigma)\ \textit{then please grant}\ \texttt{req}(\Sigma)\ \textit{else please grant}\ \emptyset$$

I_2 evaluates $\texttt{req}(\Sigma)$ and $\texttt{max}(\Sigma)$. If either `req` or `max` detects dangerous tampering to Σ, then that function will request $\emptyset$. Likewise, if `req` makes an excessive request, then the conditional ensures that the result will be $\emptyset$. Since $\texttt{run} \notin \emptyset$, the agent will then not be run by I_2. Otherwise, P_2 has requested some set α_0 of permits.

In the logic of authentication presented in [10], authorization—the *granting* of permits—is carried out using access control lists. Logically, an access control list is a set of formulas of the form $(Q\ \textbf{says}\ s) \supset s$, where the statements s are requests for access to resources. If a principal P **says** s_0, then I_2 tries to match P and s_0 against the access control list. For any entry $(Q\ \textbf{says}\ s) \supset s$, if $P \Rightarrow Q$ and $s_0 \supset s$, then I_2 may infer s, thus effectively granting the request. This matching may be made efficient if P, Q, and s take certain restricted syntactic forms.

Since we are concerned with requests for sets of permits, if $\alpha \subseteq \alpha_0$ then *please grant* $\alpha_0 \supset$ *please grant* α. Hence, a particular access control list entry may allow only a subset of the permits requested. The permits granted will be the union of those allowed by each individual access control list entry

$$(Q\ \textbf{says}\ \textit{please grant}\ \alpha) \supset \textit{please grant}\ \alpha$$

that matches in the sense that $P_2 \Rightarrow Q$ and $\alpha \subseteq \alpha_0$.

Authorization for Travel Agents We again return to our example of mobile travel agents, in order to illustrate how state appraisal functions may be used to achieve their security goals. In particular, we will stress the goals 2 and 3 of Section 2, namely flexible selection of permits and the use of state appraisal functions to detect malicious alterations.

In our first example, we will illustrate how the flexible selection of privilege may benefit the servers, in this case the airlines. Let us suppose that agents requesting full-fare, unrestricted tickets are to be given priority over those requesting restricted tickets at discount fares. Since the author of the program knows best how these facts are stored in the agent's state, he knows how to perform this test. It may thus be incorporated into the state appraisal function `max`. Since the author is a disinterested, knowledgeable party, the interpreter can safely grant a requested privilege whenever it is within $\texttt{max}(\Sigma)$. The priority treatment might consist of faster service, or access to a larger pool of seats.

The flexible selection of privilege may also benefit the sender. For instance, when the agent returns to the travel agency's trusted interpreter, it may attempt to book a ticket if sufficient information has already been retrieved. The sender may decide that she wants at least four airlines to be consulted. Thus, she writes—or selects—a permit request function `req` that, on reaching the travel agency's host, checks the number of candidate itineraries in the agent's state.If this is at least four, then she requests the `make-booking` permit. An exception-handling mechanism will send the agent out for more information if it tries to make a booking without this permit.

State appraisal may also be used to disarm a maliciously altered agent. Let us alter our example slightly so that the agent maintains—in its state—a linked list of records, each of which represents a desired flight. The code that generates this list ensures that it is finite (free of cycles); the code that manipulates the list preserves the invariant. Thus, there is never a need for the program to check that this list is finite. However, when an agent is received, it is prudent for the state appraisal function to check that the list has not been altered in transit. For if it were, then when the agent began to make its reservations, it would exhaust the available seats, causing legitimate travelers to choose a different carrier.

5 Conclusion

In this paper, we have briefly described a framework for authenticating and authorizing mobile agents, building on existing theory. Our approach models a variety of trust relations, and allows a mobile agent system to be used effectively even when some of the parties stand in a competitive relation to others. We have introduced the idea of packaging state appraisal functions with an agent. The state-appraisal functions provide a flexible way for an agent to request permits, when it arrives at a new interpreter, depending on its current state, and depending on the task that it needs to do there. The same mechanism allows the agent and interpreter to protect themselves against attacks in which the state of the agent is modified at an untrustworthy interpreter or in transit. We believe that this is the primary security challenge for mobile agents, beyond those implicit in other kinds of distributed systems.

References

1. L. Cardelli. A language with distributed scope. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages*, pages 286–298, 1995. `http://www.research.digital.com/SRC/Obliq/Obliq.html`.
2. H. Cejtin, S. Jagannathan, and R. Kelsey. Higher-order distributed objects. *ACM Transactions on Programming Languages and Systems*, 17(5):704–739, September 1995. `http://www.neci.nj.nec.com:80/PLS/Kali.html`.
3. D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, and G. Tsudik. Itinerant agents for mobile computing. IEEE *Personal Communications Magazine*, 2(5):34–49, October 1995. `http://www.research.ibm.com/massive`.

4. W. Farmer, J. Guttman, and V. Swarup. Security for mobile agents: Issues and requirements. In *National Information Systems Security Conference*. National Institute of Standards and Technology, October 1996.
5. C. G. Harrison, D. M. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? Technical report, IBM Research Report, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY, March 1995. `http://www.research.ibm.com/massive`.
6. C. Haynes and D. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9:582–598, 1987.
7. IBM Corporation. Things that go bump in the net. Web page at `http://www.research.ibm.com/massive`, 1995.
8. IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.
9. Charlie Kaufman, Radia Perlman, and Mike Speciner. *Network Security: Private Communication in a Public World*. PTR Prentice Hall, Englewood Cliffs, 1995.
10. B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10:265–310, November 1992. `http://DEC/SRC/research-reports/abstracts/src-rr-083.html`.
11. Sun Microsystems. Java: Programming for the internet. Web page available at `http://java.sun.com/`.
12. Sun Microsystems. HotJava: The security story. Web page available at `http://java.sun.com/doc/overviews.html`, 1995.
13. J. Tardo and L. Valente. Mobile agent security and Telescript. In *IEEE CompCon*, 1996. `http://www.cs.umbc.edu/agents/ security.html`.
14. C. Thirunavukkarasu, T. Finin, and J. Mayfield. Secret agents — a security architecture for KQML. In *CIKM Workshop on Intelligent Information Agents*, Baltimore, December 1995.
15. T. D. Tock. An extensible framework for authentication and delegation. Master's thesis, University of Illinois at Urbana-Champaign, Urbana, IL, 1994. `ftp://choices.cs.uiuc.edu/Papers/Theses/MS.Authentication.Delegation.ps.Z`.
16. Leendert van Doorn, Martín Abadi, Mike Burrows, and Edward Wobber. Secure network objects. In *Proceedings, Symposium on Security and Privacy*, pages 211–221. IEEE Computer Society, 1996.
17. J. E. White. Telescript technology: Mobile agents. In *General Magic White Paper*, 1996. Will appear as a chapter of the book Software Agents, Jeffrey Bradshaw (ed.), AAAI Press/The MIT Press, Menlo Park, CA.