A SECURITY MODEL FOR AGLETS

Aglets are Java-based mobile agents developed at IBM's Tokyo Research Laboratory. This article describes a security model for the aglets development environment that supports flexible architectural definition of security policies.

GÜNTER KARJOTH
IBM Research Division, Zurich Research Laboratory
DANNY B. LANGE† AND MITSURU OSHIMA
IBM Research Division, Tokyo Research Laboratory

Mobile agents offer a new paradigm for distributed computation, but their potential benefits must be weighed against the very real security threats they pose. These threats originate not just in malicious agents but in malicious hosts as well.[1] For example, if there is no mechanism to prevent attacks, a host can implant its own tasks into an agent or modify the agent's state. This can lead in turn to theft of the agent's resources if it has to pay for the execution of tasks, or to loss of the agent's reputation if its state changes from one host to another in ways that alter its behavior in negative ways.

Moreover, if mobile agents ultimately allow a broad range of users to access services offered by different and frequently competing organizations, then many applications will involve parties that may not trust each other entirely.[2] The operation of a mobile agent system will therefore require security services that implement the agreements made by the involved parties, whether declared or tacit. Thus, the agreements cannot be violated, either accidentally or intentionally by the involved parties or by malicious or curious parties not bound by the agreements.

†Since collaborating on this article, Danny B. Lange has joined General Magic, Inc.

1089-7801/97/$10.00©1997 IEEE

The security frameworks of Java and other script languages for "remote programming" such as Safe Tcl have allowed developers to make some progress toward one issue of mobile agent security—namely, the safe execution of untrusted code—through restricted environments based on sandboxing or a separated execution environment. Some current agent systems offer basic privacy mechanisms such as a secure channel between machines via encryption of agents and messages on transmission. Some offer means of authentication and integrity via the signing of agents and messages sent between hosts, again using a variety of cryptographic tools. Even fewer agent systems (Agent Tcl,[3] Telescript[4]) offer mechanisms to control resource consumption. Finally, the Mobile Agent Facility* under development at the Object Management Group will include a security model based on the CORBA security specification. However, no system at present provides a general security model.

In this article, we present our security model for the IBM Aglets Workbench,* a Java-based environment for building mobile agent applications. We detail both the security model and corresponding security architecture that represents a framework for the inclusion of security services in future releases of the AWB. This work therefore represents an additional step toward the comprehensive security model required for widespread commercial adoption of mobile agent systems to occur.

AGLETS WORKBENCH

The IBM Aglets Workbench lets users create *aglets*, mobile agents based on the Java programming language. The AWB consists of a development kit for aglets and a platform for their execution. It is based on the aglet object model, whose major elements are aglets, contexts, and messages. The Aglet Transfer Protocol (ATP) and the Aglet API (A-API) are further AWB components that define how to transport aglets and how to interface to aglets and contexts.

In this section we briefly describe these elements as far as necessary to understand the security work presented next. For more details on aglets, see Lange and Oshima,[5] available as a working draft "cookbook" on the aglets site* at the Tokyo Research Laboratory. For tutorials on aglets and AWB, see Sommers[6] and Venners.[7,8]

Aglets Object Model

Aglets are serialized Java objects that visit aglet-enabled hosts in a computer network. An aglet that executes on one host can halt execution, dispatch to a remote host, and resume execution there. When the aglet migrates, it takes along its program code as well as its data. An aglet is autonomous because it runs in its own thread of execution after arriving at a host; it is reactive because it can respond to incoming messages.

Table 1. Aglet life-cycle events and their methods.

Event	Methods: As the event takes place	Methods: After the event has taken place
Creation		onCreation()
Cloning	onCloning()	onClone()
Dispatching	onDispatching()	onArrival()
Retraction	onReverting()	onArrival()
Disposal	onDisposing()	
Deactivation	onDeactivating()	
Activation		onActivation()
Messaging	handleMessage()	

A *context* is an aglet's workplace. It is a stationary object that provides a means for maintaining and managing active aglets in a uniform execution environment where the host system is secured against malicious aglets. A *proxy* is a representative of an aglet. It serves as a shield to protect the aglet from direct access to its public methods. The proxy also gives the aglet location transparency; that is, it can hide the aglet's real location.

A *message* is an object exchanged between aglets. As mobile and autonomous objects, aglets do not exist in statically configured object structures but must instead interact with objects that might originate from unknown sources. Aglets therefore communicate by message passing and not by method invocation. Message passing allows flexible interaction and exchange of knowledge between systems.

Other agent languages, for example Agent Tcl[3] and Telescript,[4] focus on process migration, which lets an agent "leave" one machine in the middle of a loop and resume execution in the middle of that loop on another machine. Aglets, by comparison, use an event-based scheme, as in window system programming. They implement several event-handling methods, which can be customized by programmers. These methods cover all important events in an aglet's life cycle (see Table 1). For example, if an aglet is moved, it will be notified upon leaving its host and upon arriving at the new host.

An aglet is created within a context. The new aglet is assigned an identifier, inserted into the context, and initialized. The aglet starts to execute as soon as it has been initialized. The cloning of an aglet produces an almost identical copy of the original in the same context, except that the clone has a different identifier and restarts execution.

Dispatching an aglet from one context to another will remove it from its current context and insert it into the destination context, where it will restart execution. We say that the aglet has been "pushed" into its new context. The retrac-

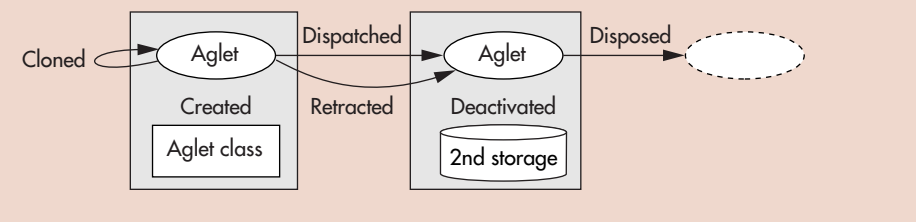

Figure 1. Aglet life cycle.

tion of an aglet will "pull" (remove) it from its current context and insert it into the context from which the retraction was requested.

Deactivation of an aglet removes it temporarily from its current context and holds it in secondary storage. Activation of an aglet restores it into a context. Disposal of an aglet halts its current execution and removes it from its current context. Figure 1 illustrates these events in the life cycle of an aglet.

Aglets communicate via messages. Each aglet can be equipped with a message-handling method that lets it react to incoming message objects sent from another (possibly remote) aglet. Message handling can be synchronous or asynchronous. A future-reply object returned by the message-sending method allows the aglet either to wait for a reply or to continue processing and get the reply later. An aglet can also multicast a message to all aglets within the same context that have subscribed to that message.

Aglet API

The Java Aglet API[9] defines the methods necessary for aglet creation and manipulation. A-API is public and therefore allows the development of platform-independent mobile agents written in the Java programming language. Aglets written to the API will run on any machine that supports it. A-API has two core classes and core interface.

The Aglet class, a subclass of Object, is the abstract base class. It defines final methods for controlling an aglet's own life cycle—namely, methods for cloning, dispatching, deactivating, and disposing of itself. It also defines methods that are supposed to be overridden in its subclasses by the aglet programmer, and provides "hooks" to customize an aglet's behavior. These methods are invoked systematically by the system when certain events take place in the life cycle of an aglet (see Table 1).

The AgletProxy class serves as a shield for aglets, protecting them from direct access to their public methods. Interaction with an aglet takes place only via its proxy. Aglets do not interact with other aglets by invoking their methods. For example, a proxy is returned on any of the following aglet creation requests:

- AgletContext.createAglet(...);
- Aglet.clone();
- AgletProxy.clone();

The context or other aglets might use several of the proxy's methods—such as clone(), dispatch(), dispose(), and deactivate()—to control the aglet. The method sendMessage() is used to send asynchronous messages to the aglet via its proxy.

An aglet uses the AgletContext interface to obtain information about its environment and to send messages to the environment, including to other aglets currently active in it. The interface provides means for maintaining and managing active aglets in an environment where the host system is secure against malicious aglets. If an aglet has access to a given context, it can create new aglets or retract remotely located aglets into the current context. It can also retrieve a list (enumeration) of proxies of its fellow aglets present in the same context.

The aglet context is typically created by a system having a network daemon that listens for aglets. The daemon inserts incoming aglets into the context. Often, a user interface component will provide a graphical or command line interface to the context. In general, any user can set up a context. Thus, an aglet network potentially includes contexts that not all users trust.

THREATS, ATTACKERS, AND COUNTERMEASURES

There are four security issues specific to a mobile agent system.[3] They are

- protection of the host against aglets,
- protection of other aglets,
- protection of the aglet from the host, and
- protection of the underlying network.

Whereas the literature discusses all of these issues (for example, see Chess et al.[10] and Farmer et al.[2]), researchers have found serious solutions only for the first two. Our security model for the Aglets Workbench also focuses on these two issues, although the model is flexible enough to accommodate eventual solutions to the latter two as well.

While developing the model, we assumed that an aglet system is subject to the fundamental threats of disclosure, modification, denial of use, misuse, abuse, and repudiation. These threats are possible not only when the aglets travel but also when they are in aglet contexts. We assumed that attackers can perform passive and active attacks utilizing aglets, aglet contexts, or other mechanisms.

Table 2 lists and briefly describes the attacks possible on aglets. A secure aglet system must provide services to counter these threats. However, there is no countermeasure if the attacker

exploits system flaws or security weaknesses such as bypassing controls, exploiting trapdoors, or introducing Trojan horses.[11]

A security architecture must therefore confine key security functionality to a trusted core that enforces the essential parts of the security policy. These parts include

- protecting aglet transfer and communication as required by the security policy,
- performing the required access control and auditing of aglet execution,
- preventing (groups of) aglets from interfering with each other or gaining unauthorized access to each other's state, and
- preventing aglets from interfering with their hosting aglet system.

Additional requirements to be met in some systems include

- allowing the use of different cryptographic algorithms,
- keeping the amount of information encrypted for confidentiality to a minimum, and
- being compatible with standard distributed security frameworks such as those of IETF,* X/Open,* and OMG.*

However, there are security requirements for agents and hosts that cannot be fulfilled.[2,10] It is impossible, for example,

- to hide anything within an agent without the use of cryptography,
- to communicate secretly with a large, anonymous group of agent platforms,
- to prevent agent tampering unless trusted hardware is available in agent platforms, and
- to distinguish an agent from a clone.

These limitations imply that an agent cannot carry its own key (or other secrets, such as a credit card number) in a form that can be used on untrusted hosts.

Moreover, it is impossible for an agent to verify whether

- an interpreter is untampered,
- an interpreter will run an agent correctly,
- a host will run an agent to completion, or
- a host will transmit an agent as requested.

Because aglets are Java objects, they have potential access to all Java class files on the host; they also rely on the security of the Java interpreter for their proper execution. Thus, aglet security and Java security go hand in hand. All the security concerns raised about Java also affect the safe execution of aglets (for example, see FAQs at JavaSoft* and the Princeton Secure Internet Programming Team.* A small local bug in the implementation of the hosting Java interpreter will affect the security of the Aglet Workbench.

Together, these limitations outline the bounds of possibility achievable by technological means only. Although legal and social controls may offer other means of protecting mobile agents, the scope of our security model is restricted to solutions achievable with standard security technology.

Table 2. Possible attacks on aglets.

Eavesdropping	Information is revealed from monitored communications.
Intercept/alter	A communicated data item, such as a Java class file, is changed, deleted, or substituted while in transit. In particular, any context visited on the aglet's itinerary could strip data added by previous contexts.
Replay	A captured copy of a previously sent legitimate aglet is retransmitted for illegitimate purposes.
Masquerade	An entity pretends to be a different entity, for example, one aglet pretends to be another.
Resource exhaustion	A resource is deliberately used so heavily that service to other users is disrupted.
Repudiation	A party to a communication exchange later denies that the exchange took place.

SECURITY MODEL

We have developed a security model that provides an overall framework for aglet security. The model supports the flexible definition of various security policies and describes how and where a secure system enforces these policies.

Security policies are defined in terms of a set of rules by one administrative authority. The policies specify

- the conditions under which aglets may access objects;
- the authentication required of users and other principals, which actions an authenticated entity is allowed to perform, and whether entities can delegate their rights;
- the communications security required between aglets and between contexts, including trust; and
- the degree of accountability required for each security-relevant activity.

An aglet might be unaware of the security policy of the hosting context and how it is enforced. If so, the user can be authenticated prior to creating the aglet; security is then enforced automatically. Some aglets will need to control or

influence which policy is enforced by the system on their behalf, but will not enforce it themselves. Others will need to enforce their own security, to control access to their own data, or to audit their own security-relevant activities.

Principals and Identities

A principal is any entity whose identity can be authenticated by a system that the principal may try to access. A principal can be an individual, a corporation, a daemon thread, or a smart card. An identity consists of a name and possibly other attributes.

Our aglets security model identifies several principals, each having certain responsibilities and interests, which are summarized in Table 3. Aglets and contexts are processes (threads) running on behalf of a user; manufacturers, owners, masters, and authorities are users imposing "roles" on aglets and contexts that reflect their organizational, functional, or social position.

Aglets. Every aglet has an identifier that is unique over its life time and independent of the context it is executing in. Its value, however, is not known before the aglet has been created and thus not easily accessible for authorization purposes. Therefore, the aglet identity includes its class name—a kind of "product" name—for authentication. The identifier might be used in policies that refer to specific instances of aglets; for example, it might indicate that a particular aglet can dispose of any of its offsprings. An aglet's product name might be used when only a certain type of aglet is meant; for example, aglets of class ibm.aglets.samples.watcher might have access to specific HTML files.

The aglet manufacturer produces a well-defined and reliable aglet. It is in the manufacturer's interest that no damage can be claimed to have been caused by a malfunctioning aglet. For its own protection, the manufacturer might define terms of liability.

The aglet owner is concerned mainly about the safety of the launched aglet. Can the returned results be trusted? Did every context execute the aglet properly? For that purpose, the owner may define security preferences, a set of rules that specify who may access/interact with the aglet on its itinerary. However, as the aglet has to rely on the context to carry out compliance, the preferences are no more than a statement of intent. Security preferences also allow the owner to limit the aglet's capabilities, for example to specify some global allowance on the maximum number of hops, CPU-time consumption, and so on.

Contexts. The context manifests the execution platform of the aglet. Its identity is the URL of the host together with a qualifier if there is more than one context. Unlike aglets, contexts are long-lived objects and thus may keep their identity—that is, their address—even after updates or complete replacements of the software and hardware that realize the context. For security-critical applications that associate trust with a specific version of the context, the identity of a context must have an attribute like the serial number of a CPU. Just as a software license can be granted to only one specific computer, identified by the serial number of its CPU, a context's serial number refers to a specific release of its software and hardware.

The context manufacturer produces a reliable context according to the A-API specification. Again, it is in a manufacturer's interest that no damage can be claimed to have been caused by a malfunctioning context. The manufacturer's specification of the context's functionality sets the basis for the context master.

The context master is responsible for the safety and security of the context and its underlying machine. A master defines the security policy for the context under its control, that is, for protecting local resources against aglets. The master is also responsible for guaranteeing that no aglet can interfere with any other aglet active in the same context if not explicitly permitted by the aglet owner.

Domains. Several things make it appealing to organize contexts into groups. For example, a context provides a certain infrastructure—general services for aglet administration (creation, activation, retraction), communication, support of audio and images—as well as specific security services such as authentication, authorization, and accounting. A single context providing all these services might be very expensive, so grouping contexts can be efficient and cost-effective. It might also easily achieve secure communication between contexts of the group if communication is local and thus protected by means of the operating system.

All domain members follow the same security policies as set up by the domain authority. In some cases the DomainAuthority and the ContextMasters of the domain members are the same principal. In other cases the DomainAuthority and the ContextMasters are different principals, as in the case of a mall provider and a set of shop owners.

A domain might correspond with an Internet subdomain, for example the set of all contexts with the address *.trl.ibm.com. It might also be the set of all contexts owned by the same master or defined by a directory or a certificate stating the context's membership in the domain.

Security Policies

All principals introduced here may define security policies, in particular the aglet owner and the context master. Thus, a secure aglet system should implement the overall effect of all security policies involved. For example, although the aglet owner might have specified that the aglet can consume up to 10 seconds within each context visited, the context mas-

ter can set a limit of 5 seconds, which will override the owner's limit.

The hierarchy of security policies defined by different principals is

AgletManufacturer < AgletOwner < ContextMaster < DomainAuthority

indicating that the domain authority sets the basic policies on the execution of aglets within a given context, which can then be refined but not overwritten by the context master, aglet owner, and aglet manufacturer.

A policy database represents the policy defined by the context master; security preferences represent the policy defined by the aglet owner.

Table 3. Principals defined in the aglets security model.

Aglet	Instantiation of the aglet program itself (thread)
AgletManufacturer	Author of an aglet program (human, company, content rating service, and so on)
AgletOwner	Individual that launched the aglet (human) or principal that has legal responsibility for the aglet's behavior
Context	Interpreter that executes aglets (process, thread, and so on)
ContextManufacturer	Author of a context program/product (human, company, and so on)
ContextMaster	Owner/administrator/operator of the context (human, company, and so on)
Domain	A group of contexts owned by the same authority
DomainAuthority	Owner/administrator/operator of the domain (human, company, and so on)

Aglet Mobility

If the aglet manufacturer, aglet owner, context manufacturer, and context master can be properly identified—for example, by their public key and with the help of a suitable certification infrastructure—the following example describes the steps it takes to create an aglet and to let it travel securely.

Before the owner can launch an aglet, the context authenticates the owner as a registered user. Within the creation request, the aglet owner defines security preferences to be applied on the aglet. When the context instantiates the aglet from the corresponding Java class, it might include information about the manufacturer, owner, and the aglet's original context—that is, about itself. This information, together with the aglet code and the owner's security preferences, forms the static part of the aglet, and will be signed by the context. Thus, any receiving context can verify the integrity of the static part of the aglet.

Aglets move when they are either dispatched to or retracted from a remote location. We use the following terminology to describe an aglet's travel:

- origin context—the context in which the aglet has been created.
- destination context—the context that receives an aglet.
- current context—the context that delivers the aglet to the receiving context.

Current and destination contexts establish a secure channel between themselves. The current context protects the integrity of aglet data by computing a secure hash value that allows the destination context to perform after-the-fact detection of tampering during the aglet's transit. Unauthorized parties can be prevented from reading sensitive information held by an aglet while it is in transit between two aglet contexts if the peer contexts agree on the use of cryptography for encryption.

For each context, a security policy describes the proper communication mechanism with any peer context. For example, although an aglet might not require any security protection for its transfer to the destination context, the destination context's security policy may lay down the use of the Secure Socket Layer (SSL) protocol with client authentication.

Access to Local Resources

When an aglet enters a context, the context receives a reference to it, and the aglet resumes execution in the new context. The aglet can also obtain a reference to the context interface. An aglet uses the context to gain information about its new environment, in particular about other aglets currently active in the context in order to interact with some of them.

Contexts have to protect themselves against aglets and aglets must be precluded from interfering with each other. The aglet context establishes a reference monitor, which gives an aglet access to a resource only if it complies with the access control policy instated by the context master. Thus a context establishes a domain of logically related services under a common security policy governing all aglets in that context.

The master of the context configures authorization policies for incoming aglets. In general, there is the following hierarchy of authorization policies:

- general level for an unauthenticated manufacturer,
- organization level for an unauthenticated owner, and
- per-aglet level otherwise.

In addition, authorization may be given with respect to computing power, occupancy level, organizational affiliation, pricing, code certification, or the type of aglet (such as a game or search aglet).

According to a security policy defined using this hierarchy, the reference monitor of an aglet context might give permission to obtain file information; to read, write, or delete local files; to connect to a network port on the origin context or to any other context; to load a library; or to create a pop-up window. These resources are taken from the Java model. In an aglet system, there are additional resources for such things as creating new aglets, cloning a specified aglet, and dispatching or disposing of an aglet.

Because an aglet carries the security preferences of its owner, it usually includes rules that govern its consent for cooperation. However, the aglet has to rely on the context for compliance. The aglet's security preferences describe who and under which circumstances the context or another aglet may dispose of, deactivate, clone, dispatch, or retract the aglet. The preferences may further define which other aglets may call which of its methods.

SECURITY ARCHITECTURE

The security architecture implements the security model by providing a set of components and their interfaces. In this section, we introduce two components of the aglets security architecture: the policy database of the context master and the preferences of the aglet owner. Because both context master and aglet owner have their own specific interests concerning what an aglet should be able to do, both may want to restrict its capabilities. Such restrictions might apply to either accessing the local resources of a context or offering services to other aglets. The policy database and security preferences therefore constitute powerful elements in introducing security into the Aglets Workbench.

Any useful mobile agent system must implement general and flexible security policies. Our model simplifies the administration of these policies by introducing the notion of roles, namely, the manufacturer, owner, master, and authority principals. In the following, we describe a language for defining policies using the concepts presented in our security model, and show how a context master and an aglet owner can use it to define their policies. The language provides named groups, composite principals (a set of principals), and hierarchical resources with associated permissions that allow the definition of high-level authorization policies. To allow fine-grained control, a security policy consists of a set of named privileges and a mapping from principals to privileges. Furthermore, the language allows the definition of black lists that disallow aglets and contexts known not to behave well.

Authorization Language

For illustration, we define the following principals:

- manufacturer—Hermes, Athena, Cronos
- owner—Semele, Leda
- master—Apollo, Hades
- authority—Zeus
- context—Olympus, Underworld
- aglet—Castor, Pollux

We use these names to simplify our discussion, but the real value of Olympus might be something like atp://www.trl.ibm.co.jp and its product name might be ibm.aglets.tahiti.Tahiti. The product name of aglet Castor might actually be ibm.aglets.samples.Writer.

Basic principals address single aglets or groups of aglets. The following are examples of basic principals:

- aglet=Pollux—denotes the aglet Pollux.
- owner=Semele—denotes all aglets launched by Semele.
- manufacturer=Hermes—denotes all aglets written by Hermes.
- context=Underworld—denotes all aglets arriving directly from Underworld.
- master=Apollo—denotes all aglets arriving directly from contexts mastered by Apollo.
- authority=Zeus—denotes all aglets arriving directly from contexts controlled by Zeus.

A manufacturer might become authenticated by a signed Java class file. A master might become authenticated by peer authentication. The use of wild cards enables the specification of groups of contexts, for example, www.*.ibm.com or *.edu.

In particular, principals that denote contexts or masters indirectly identify the context from which an aglet has arrived. By convention, when an aglet is launched, the context and master refer to the corresponding principal of the local host.

Composite principals offer a convenient way to combine privileges that must be granted to multiple principals into a single access right. Such a grouping feature considerably simplifies the security administration. Membership in a group supports the combination of various principals that should have the same access rights. For example, the following specifications combine principals of the same type into named groups and use these group names later in rule definitions:

```
GROUP AssociationOfManufacturers=Hermes,Athena
```

This rule indicates that group "Association Of Manufacturers" consists of Hermes and Athena.

```
Cronos IS_MEMBER_OF Titans
```

This rule adds Cronos to group Titans.

Three other constructors denote set difference (EXCEPT), set differences (OR), and set intersection

(AND). Set union is useful for handling exceptions, such as a privilege that should be given to a group except for a certain user.

The following are examples of these constructions:

- owner=Leda OR context=Underworld—any aglet owned by Leda or arriving from context Underworld.
- owner=Semele AND context=Underworld—any aglet owned by Semele and arriving from Underworld.
- manufacturer=AssociationOfManufacturers EXCEPT manufacturer=Titans—any aglet written by any member of "Association Of Manufacturers" except those written by members of group Titans.
- owner=Semele EXCEPT manufacturer=Cronos—any aglet launched by Semele but not written by Cronos.

Privileges

Privileges define the capabilities of executing code by setting access restrictions and limits on resource consumption. A privilege is a resource, such as a local file, together with appropriate permissions such as read, write, or execute in the case of the local file. Our security architecture currently considers the following resource types:

- File—files in the local file system
- Net—network access
- Awt—the local window system
- System—any kind of system resources, such as memory and CPUs
- QoP—quality of protection
- Context—resources of the context
- Aglet—resources of the aglet

Resources are structured hierarchically. Thus, permissions can be given to a set of resources or even to a complete resource type, like universal file access. An example with a simple hierarchy is the resource type File:

- File—all files
- File/tmp/sample.txt—the file /tmp/sample.txt

Net access is a more elaborate resource. Our authorization language lets you distinguish among different protocols (for example, TCP and HTTP) and select ports or port ranges to build resources:

- Net—any kind of networking
- Net TCP—any kind of TCP connections
- Net TCP host—TCP connections to host
- Net TCP host port—TCP connections to host but only on port

Each resource also has a corresponding set of permissions.

URLs FOR THIS COLUMN

*MAF • www.genmagic.com/agents/MAF
*Aglet Workbench • www.trl.ibm.co.jp/aglets/
*IETF • www.ietf.org/
*X/Open • www.opengroup.org/
*OMG.• www.omg.org/
*Security FAQs • java.sun.com/sfaq and www.cs.princeton.edu/sip/java-faq.html
*DARPA Workshop on Foundations for Secure Mobile Code • www.cs.nps.navy.mil/research/languages/wkshp.html

The permissions for networks are send, receive, any, connect, and accept.

The services provided by the aglet context are also subject to control. The context provides methods to create, retract, and activate aglets; to send or receive messages to/from other aglets; to obtain aglet proxies, the hosting URL, audio clips, and images; and to get or set the properties of the context. The following are example privileges:

- Context AGLET retract—the aglet can retract any aglet in the context.
- Context AGLET owner=Leda retract—the requester of method retractAglet can retract the specified aglet if the owner of the retracted aglet is Leda.
- Context PROPERTY origin get—the aglet can retrieve property origin of the context.
- Context MESSAGE subscribe—the aglet can subscribe to messages.

Combining resources with permissions, privileges are defined as follows:

- File/tmp read,write—the aglet is allowed to read and write from tmp.
- Net TCP Underworld 930-933 NOT connect—the aglet cannot connect to context Underworld using TCP on ports 930-933. (Note that this privilege expresses negative permission.)
- System LIBRARY ibm.db2.info—the aglet can load library ibm.db2.info.
- System MAX_MEMORY 12—the aglet may not allocate more than 12 Mbytes of memory.
- System MAX_DISK_SPACE 200—the aglet may not consume more than 200 Kbytes of disk space.
- AWT Top_level_windows 1—the aglet can create one top-level window.

Our authorization language also introduces a special permission called enter, which allows an aglet to enter the context if granted. Used as a negative permission, it can exclude

certain aglets from executing in that context (again, the idea of a black list).

Context Master Policy Database

The context master defines the security policy for aglet contexts under its control. This policy defines the actions an aglet can take. In the policy database, the context master combines principals that denote aglet groups and privileges into rules. The syntactic form of a rule is

<label>:<principal> -> <privileges>

When an aglet matches multiple principals, we say that a "consensus voting rule" combines the policies for those principals. In other words, a negative rule rejects the request. The contents of a policy database might then look like this:

```
TRUSTED:
manufacturer=Athena OR master=Hades ->
File /tmp read,write
Net TCP Underworld accept
Top_level_windows 3
Aglet owner=Leda retract
GUEST:
manufacturer=Hermes ->
Net message Olympus receive
Top_level_windows 1
Aglet Property get,set
REJECT:
manufacturer=Hermes,Titans ->
Context NOT enter
```

Note that none of the aglets mapping into the Reject group will be allowed to enter the context because they do not have the necessary enter privilege.

Aglet Owner Preferences

The aglet owner has the opportunity to establish a set of security preferences that will be honored by the contexts the aglet might visit. Preference combines context groups and privileges into rules. The syntactic form of a rule is

<label>:<context_group_definition> -> <privileges>

The following list defines the set of methods on aglets that an owner can restrict:

- clone/deactivate/dispatch/retract/dispose
- get AgletClassName/AgletContext/CodeBase/Identifier/Itinerary/MessageManager/Property/PropertyKeys/Text
- send Message
- set Itinerary/Property/Text
- subscribe/unsubscribe (all) messages

These actions can be requested by the aglet itself or by other actions via the AgletProxy. The following are examples of security preferences:

- context=Olympus EXCEPT master=Apollo -> ITINERARY set—the aglet's itinerary might be changed at context Olympus but only if this context is not mastered by Apollo.
- master=Hades -> MESSAGE welcome subscribe—at all contexts mastered by Hades, the aglet might subscribe to messages of kind welcome.
- -> aglet=Pollux OR owner=Leda AGLET dispose—at any context the aglet might only be disposed of by aglet Pollux or any other aglet owned by Leda.

Allowances are preferences dealing with the consumption of resources such as CPU time or memory. They can be local, concerning only the current context, or global and thus apply to the set of all hosts visited. A global allowance at the time of creation puts overall limits on the aglet's action over its lifetime, effectively limiting its owner's liability. Aglets can also form groups sharing a common allowance. The allowance defines a maximum age or size, and indicates whether new aglets can be created. In the case of aglet creation or cloning, the allowance must be shared.

CONCLUSIONS

Like any other downloadable and executable code, mobile agents are a potential threat to a system. But they are also exposed to threats by their hosting system, a situation not currently dealt with in traditional security systems. It is our belief that applications based on aglets will be widely accepted only if users are convinced that security services can cope with both kinds of threats.

Our security model for aglets is a first step toward alleviating these threats. The model clearly defines the principals within an aglet system with respect to their responsibilities (liabilities) and interests. The model explains how aglets migrate, and depicts their access to local resources. Thus it serves as a reference for a corresponding security architecture. We introduced two elements of the security architecture—the policy database and owner-specified preferences—and demonstrated how these elements control security-unaware aglets.

Our current work addresses the aglet security API that will enable aglet application developers to enforce their own security—so that an aglet can, for example, control access to its own data or audit its own security-relevant activities. Such security-aware aglets could implement, say, the secure KQML, as proposed by Thirunavukkarasu et al.,[12] using the offered API primitives. The API design takes into account the security features added to the Java Developer's Kit Version 1.1 and subsequent versions. In particular, protection domains and a uniform way to access user identities that

were established in different ways have been proposed.[8] This may simplify the implementation of the aglet security API.[13] However, to prevent denial-of-service attacks and thus to implement the observance of allowances, the context must monitor resource consumption. This may not be possible without also changing the Java virtual machine.

Although our authorization language is already quite expressive, we will extend it to support contextual information, such as aglet history and time, in access decisions. For example, a policy that allows network access only if the aglet has not previously accessed the file system certainly allows network permission to be given to a larger group of aglets.

We have not resolved how to protect an aglet's internal state against snooping and tampering, and a generic solution to this problem is still a very challenging research topic. However, there are security mechanisms today for limited mobile agent applications, and more will be developed soon. Proposals for such mechanisms were discussed recently at a DARPA Workshop on Foundations for Secure Mobile Code.* Our strategy is to provide a well-defined and rich set of security services within the Aglets Workbench that will enable the implementation of these mechanisms to better protect an aglet from a malicious host. ■

REFERENCES

1. J.J. Ordille, "When Agents Roam, Who Can You Trust?" *Proc. First Conf. on Emerging Technologies and Applications in Communications* (etaCOM), http://etacom.org/, May 1996.
2. W.M. Farmer, J.D. Guttman, and V. Swarup, "Security for Mobile Agents: Issues and Requirements," *Proc. 19th Nat'l Information Systems Security Conf.* (NISSC 96), 1996, pp. 591-597.
3. R.S. Gray. "Agent Tcl: A Flexible and Secure Mobile-Agent System, in *Proc. Fourth Annual Tcl/Tk Workshop* (TCL 96), 1996.
4. J. Tardo and L. Valente. "Mobile Agent Security and Telescript," *Proc. IEEE CompCon 96*, IEEE Computer Society Press, Los Alamitos, Calif., 1996.
5. D.B. Lange and M. Oshima, *Java Agent API: Programming and Deploying Aglets with Java*, to be published by Addison-Wesley, Fall 1997; a working draft, "Programming Mobile Agents in Java, is available at http://www.trl.ibm.co.jp/aglets/whitepaper.htm.
6. B. Sommers, "Agents: Not Just for Bond Anymore," *JavaWorld*, http://www.javaworld.com/javaworld/jw-04-1997/jw-04-agents.html, April 1997.
7. B. Venners, "The Architecture of Aglets," *JavaWorld*, http://www.javaworld.com/javaworld/jw-04-1997/jw-04-hood.html, April 1997.
8. B. Venners, "Solve Real Problems with Aglets, a Type of Mobile Agent," JavaWorld, http://www.javaworld.com/javaworld/jw-05-1997/jw-05-hood.html, May 1997.
9. D.B. Lange, "Java Aglet Application Programming Interface (J-AAPI)," white paper–draft no. 2, http://www.trl.ibm.co.jp/aglets/aglet-book.html.
10. D. Chess et al., "Itinerant Agents for Mobile Computing," *IEEE Personal Communications*, Vol. 2, No. 5, Oct. 1995, pp. 34-49.
11. D. Chess. "Things That Go Bump in the Net," http://www.research.ibm.com/massive/bump.html, 1995.
12. C. Thirunavukkarasu, T. Finin, and J. Mayfield. "Secret Agents: A Security Architecture for the KQML Agent Communication Language," *Proc. Intelligent Information Agents Workshop* held in conjunction with Fourth Int'l Conf. Information and Knowledge Management CIKM 95, Baltimore, Dec. 1995.
13. L. Gong. "New Security Architectural Directions for Java," *Proc. IEEE CompCon 97*, IEEE Computer Society Press, Los Alamitos, Calif., 1997, pp. 97-102.

Günter Karjoth is a research staff member at the IBM Zurich Research Laboratory where he is currently working on access control issues in distributed object systems. His research interests are in distributed systems modeling, validation, and implementation. He received his Diplom in Computer Science and his PhD from the University of Stuttgart, Germany.

Danny B. Lange recently became director in the Agents Division at General Magic, Inc. Prior to joining General Magic, he was a visiting scientist at IBM Tokyo Research Laboratory, where he invented the Java aglet and was the chief architect of IBM's Aglets Workbench. The work reported in this article was performed while he was at IBM. Lange's research interests include mobile network agents, object-oriented frameworks, and object visualization. He received an MS and PhD in computer science from the Technical University of Denmark.

Mitsuru Oshima is a researcher at IBM Tokyo Research Laboratory. He is the head implementor and co-designer of the Aglet API. His research interests include object-oriented frameworks, component architecture, and distributed objects. He received his MS in control engineering from the Tokyo Institute of Technology, Japan.

Readers may contact Karjoth at IBM Research Division, Zurich Research Laboratory, Säumerstr. 4, 8803 Rüschlikon, Switzerland, gka@zurich.ibm.com.