Security in the JavaSeal Mobile Agent System: a Position Paper

Bryce, Ciaran

Security in the JavaSeal Mobile Agent System: a Position Paper

Ciarán Bryce

Abstract

JavaSeal is a Java-based mobile agent system currently being developed in the context of the ASAP project. This paper offers a position statement on the requirements for JavaSeal's security infrastructure, and describes on-going and future work. Security in JavaSeal relies on two features: 1) the set of agents in the system is hierarchically structured: a parent agent intercepts the messages that its children exchange and can suppress them for security reasons, and 2) agents and their security policies are Java programs, and Java's strong typing semantics is the starting point for proving security properties of these programs.

1 Introduction

Mobile agent technology is recognised as being sufficiently flexible for constructing internet applications [32]. The ability to exchange autonomous computations between sites means that a user can work in an asynchronous fashion: he logs on, sends out his agent, logs off and then comes back later to retrieve the results brought back by the agent. Mobile agents can also minimise use of the network since client-server interactions may take place at the server. At the same time, mobile agents pose security worries that have to be addressed.

In the context of the ASAP project, an agent platform called JavaSeal is being developed[1] [33]. JavaSeal is a Java package that allows programmers to define agents and that provides mechanisms for moving and securing agents. An agent is an isolated set of objects with its own threads. A key feature of JavaSeal is that agents are hierarchically structured: an agent may contain several sub-agents inside of it. A parent agent is responsible for moving its children to other agents, and also for enforcing security constraints on its children by intercepting and verifying the messages that its children exchange.

Our goal in this position paper is to describe some of the components needed for JavaSeal's security infrastructure, to outline the design choices for these components, and to describe our on-going and future work on this infrastructure. The main elements of the infrastructure include tools for isolating agents from one another, for controlling messages sent between agents, and for verifying that the parameters passed in an agent communication do not leak sensitive information. We consider the kinds of security policy that are meaningful in a mobile agent context, and how they influence the design choices for the security infrastructure. It will become clear that proving security properties within the infrastructure relies heavily on the hierarchical structure of JavaSeal agents and on Java's strong typing semantics.

1. *Agent Systems, Architecture and Platforms* project; SPP-ICS 5003-45335.

The plan of this paper is the following. Section 2 reviews the main features of the JavaSeal system - sealed objects representing agents, method dispatch for agent communication, and an agent hierarchy for controlling agent behaviour. Section 3 describes how agents are isolated from each other. Isolation ensures that the only means for communication between agents is the method dispatch mechanism; covert channels based on shared resources must be removed. Section 4 describes the kinds of security policy that are specified for JavaSeal agents, and mechanisms to enforce these policies. Section 5 concludes the paper.

2 The JavaSeal Mobile Agent System

JavaSeal is an agent platform being developed within the framework of the ASAP project. A key goal of the project is to propose clean mobility and security semantics for the Java language and its virtual machine. The specification of the system stems from a higher-order calculus called Seal that extends the pi-calculus with abstractions for seal processes and their movement [34]. As a first step, a JavaSeal prototype has been developed from this specification that provides mobility and security for threaded Java objects [33]. The system comes as a package, so there are no changes to the Java Virtual Machine and a pre-processor, at most, is needed at the language level.

An agent in the JavaSeal system is simply an instance of a furnished class *seal* (short for sealed object). A seal encapsulates a set of Java objects, classes and threads. A user programs an agent by sub-classing the class seal. The seal class and its helper classes form the JavaSeal kernel; the methods of the base seal class are used by ordinary seals to access the kernel. The kernel furnishes mechanisms for isolating seals from one another and from the JVM, as well as a seal mobility mechanism. Isolation is got by having each seal assigned its own class-loader; JVM typing rules prevent an object of one seal (class-loader space) from aliasing an object of another. A seal is also the unit of mobility in JavaSeal: a parent can ask the kernel to stop one of its sub-seals, to convert it to a data stream, and then to send it to another seal in which it can be re-started.

The JavaSeal kernel implements inter-seal communication by dispatching method calls to the called seal on behalf of the calling seal. The calling thread places its request in a queue and blocks; a thread in the called seal can then process the request, place the return value in the queue and wake the calling thread up. Parameter objects in a method dispatch are transferred by deep copy - each parameter object and object reachable from the parameter object is copied and converted into a data stream, the stream is transferred to the destination seal where the object copy is constructed. Java serialisation is currently used to implement this transfer. Deep copy is used for two reasons. The first is that it avoids type violations: each seal has its own loader space and JVM typing prohibits an object from being referenced by an object of another loader space. Since a Java serialised object does not possess class loading information, it can be passed between seals without a violation occurring. The second advantage of deep copy is that it prevents an object in a seal from ever being aliased by a malicious seal [35].

JavaSeal is *hierarchical, language-based,* and uses a *policy-less* kernel. The kernel is *policy-less* in that it does not specify a security policy governing the exchange of messages between seals, nor does it specify what variables of a seal should be saved before the seal is moved. Each seal can have its own policies implemented for this.

The JavaSeal kernel is *hierarchical* in that a seal can contain nested children seals in addition to application Java objects. A seal can only directly name its parent and immediate children; seals must use an application level naming scheme to reference seals in other places of the hierarchy. On account of this, communication between two children seals of a seal is done by the source child doing a method dispatch to the parent, who can then do a method dispatch to the second child. This model generalises to all seals; a message is first propagated to the lowest common parent of two communicating seals, and from there propagated down to the destination seal. A seal moves as a result of its parent telling it to move. Before moving, a seal is wrapped up in a data stream along with its own children seals. The data stream containing a moving seal is transferred as a parameter in a method dispatch.

A seal acts as a secure execution environment for its sub-seals. It can implement a local naming scheme, enabling child seals to name and communicate with each other, and a security policy controlling the messages exchanged between its children. A further level of security stems from a JavaSeal seal name only having a meaning in the context of the seal that contains that name. The primary advantage of this approach is that a name that "escapes" an environment is meaningless since the new environment has no way of binding that name to the object that was denoted.

Finally, the JavaSeal system is *language-based* in that agents are written in Java, and security policies are implemented as library-level classes imported as Java code, or as Java-based sub-seals, into a seal. This means that security properties are specified and reasoned about using Java language semantics. For instance, type inference techniques can be used to answer queries such as "does agent possess object (reference) of type password" [2] or "is output of method influenced by value of private key?" [2].

The importance of the JavaSeal hierarchy and Java's strong typing semantics to the security infrastructure will resurface in the next two sections when we outline elements of the infrastructure. In particular, section 3 looks at how seals are isolated from one another in such a way that effective communication is obliged to use the method dispatch technique. Section 4 then considers ways of binding security policies to method dispatches so that inter-agent communication can be securely controlled.

3 JavaSeal Security Domains

The crucial requirement for JavaSeal is that it isolate agents: there must be no way for agents (i.e., seals) to communicate except through the kernel-provided method dispatch mechanism. Applications may enforce security policies that control the messages that agents dispatch, but these policies are only effective if agents cannot surreptitiously exchange information via some other channel in the system, e.g., via a shared file.

We mentioned that the approach to isolating seals is to assign distinct class loaders to seals - the JVM loading and typing rules guarantee that no object of a class loader space may reference an object of another class loader space. In this way, user seals (which must be classes of the package seal.usr) may never reference an object whose class belongs to seal.usr though which possesses a distinct loader. More precisely, for any variable x, let type-s(x) denote its static type (the class associated with variable x in its declaration e.g., for Thread t; type-s(t)=Thread). Let type-d(x) be x's dynamic type (the class of the object currently bound to x), let class(x) be the class structure for a class x, class-s(x) be the class loader for loading class(x) in current context and finally let class-d(x) be the class loader that loaded the class of the object bound to x. The assignment a=b is legal if type-d(b) <= type-s(a) ("standard" run-time type check), and if class-s(a) = class-d(b) (the class loader of class a in the current context must be the same as that which loaded the object referenced by b) [22].

This rule hides two complications in JavaSeal regarding loading. First, the base seal class is loaded by the system loader (as are all classes in seal.sys) and a user seal is a sub-class of seal.sys.Seal. Thus, an assignment in the kernel "Seal s = (Seal)mySeal" where mySeal is loaded by a private loader is legal because of the cast. An error in kernel code could mean mySeal being assigned to a seal variable declared in another seal. For this reason, kernel code (in seal.sys.*) must be controlled. Second, seal loaders do not load their own versions of classes in java.* or seal.sys.*, rather they use the class structures loaded by the system loader. As pointed out in [35], class variables in these classes can be used to exchange information. In other words, these variables and the static variables that exist in the JVM runtime (in java.lang.* etc) can be exploited as covert channels.

Two issues need to be addressed as a consequence. First, it is important to clearly identify the set of shared (static and non-final) variables SV in the JavaSeal kernel, to make this set as small as possible, and to prove that only variables in SV can possibly be shared by different seals. Second, we must quantify as best as possible the effects of sharing, that is, try to give precise statements about the existence of covert channels through the variables in SV. This section looks at these two issues. It will become clear that techniques needed are standard program proving techniques, and that much is facilitated by the fact that JavaSeal is built using a strongly typed language like Java.

Isolation of seals: typing and loading

The only variables that should be shared between seals (the set SV) according to the JavaSeal specification are the static variables of classes loaded by the system loader (classes in seal.sys and java.*). This is a proof requirement for the JavaSeal loading process. We say that any aliasing of object b by variables a1 and a2 of different seals is only legal if b's class is in C, where C is the set of classes loaded by the system loader, or else if b is a variable of a basic type within Seal.java.

To prove this security requirement we must verify the loader code (seal.sys.SealLoader) and seal creation code, in order to trace the assignment of seal objects and the use of class loaders during JavaSeal kernel operations. This requires a formalisation of the JVM loading procedure so that one may reason about the set of classes present and the class loaders for these.

Luckily, the JVM loading and typing rules have been formalised in [17]. First presented are basic rules capturing the relations between a class, its class file and class loader, between a class and its super-class, and between classes and members with different protection visibility. Formulating the JVM state in terms of the set of loaded classes, rules are then given to describe the semantics of linking and loading classes. The semantics of the Java byte-code operations, e.g., new, invokevirtual, putstatic are then specified using link and load formulations of how the JVM class state alters. Using the basic JVM rules, one can reason about the visibility of class members for any given class. This semantics is a clear starting point for defining the set SV from the loading process, and from this, we hope to prove that variables shared must be in the specified set SV.

Even though we can identify the set SV, we must be sure that these variables are not the source of serious covert channels in the system. To make the analysis easier, the set SV has been minimised by reducing the set of classes C that can be loaded into a seal. In effect, a seal loader has a list of classes that it is allowed to load [33].

Control of sharing: covert channel analysis

We distinguish two forms of covert channels: implementation-oriented and functional. Implementation oriented channels are those based on static variables within the runtime and whose existence is purely due to some implementation decision. For instance, a method m1 that stores a parameter in a global variable x which can be read by a method m2 independently of the system specification is an implementation oriented covert channel.

Functional covert channels are those that are inherent in the system specification. A typical example of a functional covert channel is the inter-dependence of the Java *notify()* and *wait()* methods, where threads can use object lock variables to signal information to each other. The problem is that locking is essential in the JVM for concurrence control, so we cannot eliminate the channel. The best one can do in this case is to specify that threads within distinct seals cannot share locks, but that threads of the same seal may.

There are two kinds of variables in SV that we are concerned with when analysing covert channels: 1) *seal* variables are those static variables of the JavaSeal kernel classes, and 2) *system* variables are the static variables in the JVM classes. The JVM is represented by the classes of the java.lang package and other packages; these classes are written in Java and C. The presence of C complicates analysis. For instance, one cannot assume pure Java static variable semantics since the visibility rules can be broken in the C part - a class variable of one class may be accessed from the C code of a different class. For simplicity therefore, we denote the set of system variables in the following by the JVM methods that seals may use to access the run-time, e.g., *new()*, *clone()*.

To ensure that there is no sharing of seal static variables between user seals, it must be shown that the value that a seal S1 reads from a static variable is independent of any write to that variable done by another seal S2. One can formalise this by saying that for any variable for which *r()* is the method used to read and *w()* the method to write, and where *ri()* denotes the execution of *r()* by seal Si, then

|- {P} *wi(); ri()* {Q} => {P} *wi(); wj; ri()* {Q}

In other words, seal Si is unaffected by writes by seal Sj on the variable since the post-condition (which includes a value for the output of the read command) does not change. This definition is a reformulation of non-interference, a well-known security property, which expresses that the output seen by a low-level user cannot be influenced by the data contained within a sensitive object.

Programme analysis techniques have been developed for reasoning about flows of information between variables [2]. The main problem with these works is that they are too pessimistic: they are unable to distinguish between harmful and harmless information transmission. Sharing must exist in the JavaSeal kernel to implement method dispatch for example: our goal is to ensure that this sharing cannot be exploited as covert channels. What is needed in a context where shared variables must exist is a more semantic treatment. This is the essence of security in the JavaSeal system. Since the number of shared variables have been reduced to a manageable size, and since the kernel is written in Java, we can feasibly apply a more semantic treatment for reasoning about leakages from the variables.

Consider an example from seal.sys.Seal. This class has a static variable *index* which is used to assign unique indices to external references on a machine, and which is then incremented on each creation. The seal operation *op()* that updates this contains simply the code (index++; return index); *op()* fulfils the role of both *r()* and *w()*. The safe sharing rule is broken since

{index = I0} *op1;op1* {index = I0+2}, even though {index = I0} *op1;op2;op1* {index = I0+3}

This means that a seal could signal information, e.g., the value of a variable *x*, to another simply by creating *x* external references. We can close this channel simply by generating the new value of the index to be a random number, independent of the former value: the post-condition is {index=rand()}.

Reducing functional covert channels in the JVM also requires proving that a write by one seal does not interfere with a read by another. For instance, the execution of *wait()* within one seal cannot ever be influenced by a *notify()* executed in another seal. To reason about this, we would need to reason about object locks - this is feasible given since these are byte-code instructions with well-defined locking semantics. Another example problem arises for variables such as *new()* and *clone()*. A *new()* executed by one seal increases the likelihood of a the *clone()* executed within another seal returning an out of memory exception. On the one hand, treatment of this channel needs a security model for handling resource control, which we do not have. On the other hand, it at least shows how the well defined JVM semantics do simplify pinpointing covert channels in JavaSeal.

4 JavaSeal Security Policies

Security in agent systems is obtained by regulating the information exchanged between and used by agents - by controlling the services used by each agent as well as the interactions between agents. By controlling covert channels, agents in JavaSeal primarily communicate through method dispatch and so security checks are implemented at this level.

The security policy is the unit linked to the communication mechanism and which verifies that each communication is legal, and perhaps take supplementary actions such as audit and authentication. This can lead to more elaborate communication schemes where the policy needs to contact a password server or some other trusted third party. In JavaSeal, a policy is typically called by a seal when it is contacted, and by a parent seal that wants to verify the legality of a message that it reroutes from one child to another.

The goal of this section is to discuss security policy issues in the JavaSeal mobile agent context. The section begins with a look at some design questions for security policies. We then mention some simple components that are currently being used for a simple security infrastructure for JavaSeal, based on access control lists for authorisation and *credentials* for authentication. We consider police agents, ways that agents themselves can help provide security for agents. We then describe a way of controlling the parameters that agents exchange on a method dispatch so that information is not accidentally leaked in a deep copy. The section terminates with a look at some issues related to the programming of security policies for a given application.

4.1 Elements of Security Policy

At a basic level, a security policy can be programmed to fulfil several roles, e.g., 1) audit messages exchanged between agents and the services used, 2) control each access to a server or resource made by an agent, 3) verify that any unit passed as a parameter in a method dispatch does not accidentally contain a sensitive object, or 4) verify the identity of an agent that makes a request (authentication). In the context of the JavaSeal system, Java-based components for these security aspects are being designed. This paragraph considers some basic requirements for these components.

The primary requirement for any system security infrastructure is that one be able to ask the system questions about security and receive precise answers. One might ask "what rights does agent A possess?", or "if agent A receives right R, then what resources can it use on my site?", or "can I send agent A secretly to site S - does site S share a secret (key) with me?". In essence, the security infrastructure requires a *formal model* for reasoning about the effects of protection mechanisms. The challenge in JavaSeal is to define protection mechanisms that permit an expressive range of policies to be expressed in a mobile agent context, but which allow easy calculation of answers to questions such as the above. The system must be able to give some security guarantees to users. The trade-off between expressive security mechanisms and being able to reason is traditionally difficult in systems: for instance, calculation of the access state for the simple access matrix possesses non-decidability constraints [14].

A first requirement regarding the kinds of security policy is that they be *group-based* and *task-based*. A group-based policy is one where the users of the system are classed into a finite set of groups, and access to a resource is granted depending on this group. The necessity of groups comes from there being are too many users in the internet system for any given user to identify individually, and so he can only ask a user to furnish some class of identity and assign access rights based on this class. In Java for instance, an applet is assigned the group 'non-trusted' or 'trusted' depending on its source. One would nevertheless like to model a larger number of groups in the system, so that one can have more selective access controls, e.g., access is determined on basis of agent being sent by groups JavaSoft employees, SBS clerks.

To support group-based policies, agents must possess a proof of group membership, or *credentials* that allow the receiving site to attribute a group to the agent. An agent's credentials can be a digital signature, or a pointer to its byte-code so that the receiving seal may run type checks; the latter is the approach taken in Java.

Policies are *task-based* in that the access rights allocated to a user or group not only depend on the identity of the user/group, but also on the task that its agent is fulfilling. For instance, a user may be the program chair of a conference committee, in which case his agents may read submitted paper agents and assign them to reviewers. However, the PC chair should not be able to use an agent that is used for say internet browsing to read paper submissions. An agent's credentials should also be usable to indicate the task that the agent is fulfilling. This would give much dynamics to the infrastructure: agents are autonomous entities that can be contracted or delegated work by other agents; credentials encapsulate the authority that contracted agents require.

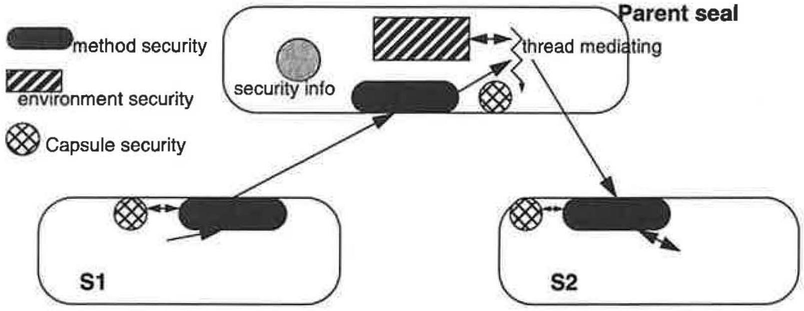
Another requirement for the security framework is *reciprocity*. Just as sites must protect themselves from mobile agents, agents must be able to protect themselves from hosts. An agent's state is exposed to the underlying system, and so a malicious site may steal or destroy any secret agent information. A general approach to this is to have a site furnish its own credentials to an agent before the agent arrives; in this way both the agent and the site can exchange credentials, and thus negotiate the agent's arrival at the site.

Another feature of security in JavaSeal is the *dynamic composition* of security policies - the security policy in place within a parent seal environment can vary over time and might not be known in advance. For instance, every electronic commerce transaction is subject to legal guidelines which form part of the *mandatory* security policy components governing agent interaction [36]. These components can differ depending on the location at which the communication occurs. This is important since it limits the amount of verification that can be done at agent compile-time. A second aspect of policy dynamism stems from the interacting agents being *autonomous*. Each agent has its own security requirements: it needs to be able to access certain resources and to protect itself from accesses by other agents. In the PC example, a submitting author requires that the contents of his paper (agent) remain confidential except to the PC - the PC members have agents that require the right to read paper agents. This means that one should be able to calculate the real security policy in place within a seal - for instance, enumerate the effective set of permitted accesses that an agent has at some time. This particularly means that the security model must specify the effects of combining policy components [9][3].

A final requirement is that a protection mechanism needs to be *expressive*, that is, one would like to be able to enforce many differing user security requirements. One way to achieve this expressiveness is by using a Java programme to represent the policy in the parent seal that controls messages exchanged between children seals [19]. Another is to use the agents themselves to implement aspects of security policies; we return to these two aspects later on in this section.

4.2 Example Policy Framework

Security is a crucial part of the JavaSeal method dispatch mechanism. Three (Java-class based) mechanisms are involved - a *method* security component, an *environment* security component, and a *capsule* security component. The method security component is invoked by a seal each time that it sends or receives a method, and protects the seal from other seals. The environment security component is used by seals whose principal purpose is an environment that coordinates and effects interaction between sub-seals; this policy component enforces security on the exchanges between children. The capsule security component is used to verify that no sensitive objects are accidentally included within a parameter graph in a method dispatch; capsule security is looked at later. The following figure outlines the main elements.



A method security component is called 4 times during a method dispatch, and uses the following Java interface:

```
public interface MethodSecurity{
public boolean sendCallMediate(Handle Caller, Object Message) // invoked by caller
public boolean receiveReturnMediate(Handle Caller, Object Message) // invoked by caller
public boolean receiveCallMediate(Handle Caller, Object Message) // invoked by callee
public boolean sendReturnMediate(Handle Caller, Object Message)} // invoked by callee
```

A security policy implementing this interface can be called during each step of a method dispatch. The first security method (sendCallMediate) is invoked when a method is sent - this is primarily used to verify the capsule security of the transferred parameters. It can also be used for byte-code verification of seal code when the method call is to the seal's createChildSeal() method - in this case the seal class files are a parameter to the method. The second function, receiveReturnMediate(), would typically be used to audit the execution of a method request, and to verify the parameters returned. The method receiveCallMediate() is used to determine if the incoming method call should be serviced (access control); one might also log the call in a log or take supplementary security actions. The final call, sendReturnMediate() is called before returning from the inter-seal call and is also used to verify capsule security by calling the capsule security component.

We currently use a Unix-like ACL mechanism for method security. This choice was made for its simplicity; though there is generally a trade-off between a simple and expressive security mechanism, the tendency is now towards simplicity, see SDSI for instance [26]. In any case, the

ACL can always be replaced in a seal by a Java class that encodes a more expressive mechanism. A seal's ACL contains access rights for seals and groups. Access rights are currently of the Unix rwx form. The execute right is a single bit that determines if the calling seal may execute a method or not. The read and write rights are useful for finer control on the methods executed, as well as for enforcing some information flow control: we need not only to control attempted accesses between domains, but also the transfer of information. Thus, when a seal arrives, the hosting seal determines a group for that seal based on its credentials. This group has access rights assigned in the ACL object; these rights can be altered using *chmod()* and *chgrp()* methods.

One role of a seal is to act as a secure environment for its sub-seals: recall that any message exchanged between two seals must traverse the common parent seal. The role of the environment policy component is to control communication between siblings of the enclosing seal. This component respects the interface verify_access(Handle callingSeal, Handle callerSeal) and in JavaSeal currently employs a simple access matrix.

The environment security component can be used to implement a spectrum of policies, and one might even represent these policies in special sub-seals. A further crucial role which the environment policy may be programmed to fulfil is audit. Since the environment policy receives a copy of all messages exchanged between sub-seals, it can record messages that can later be used as proof of delivery - this is very important in electronic transactions [36]. As another example, if a class does not agree on the outcome of a security decision, it may send a message "protesting" to the parent. This facility is useful where each agent has its own requirements to be respected, and forms the basis for supporting autonomous policies. The point is that using Java to encode the mechanisms leaves much-needed flexibility for the policies.

The model outlined is quite flexible even though still minimal. It controls mobility in a uniform way since moving agents (serialised seals) are transferred as parameters to method dispatches, and so their movement is controlled just as any other datum. Likewise, credential object exchange for authentication is implemented using method dispatch; this allows reciprocity to be implemented by a visiting agent and site seal exchanging credentials before the agent arrives and both agreeing that the visit can take place.

4.3 Credentials

Up to this point, we have not given any particular explanation to the structure of a seal's credentials, except to say that it is a Java class and a data type used by a hosting seal to assign a group to visitor seals. This is intentional - credentials are programmable to contain anything that a seal uses to identity itself or its task, or that the receiving seal might require to attribute it some level of trust. In practice, credentials might simply contain an ASCII name, a shared key or digital signature, or a pointer to the seal's byte-code so that the receiving seal can run a verification programme on the code. Assigning groups to agents by running program verification techniques such as type-checking or sand-boxing is common in Java-based agent systems, and is feasible due to the strongly-typed semantics of Java [29]. The exact format of any credentials class is thus an application level detail.

As another example, credentials could contain multiple signatures where the credentials are signed by the development environment that furnished the seal, by the storage sites that the seal passes through, as well as by the responsible user. This is important in the context of an agent environment where one typically wishes to know exactly where an agent came from before permitting it to enter one's site. One might assign an agent a higher level of trust if all of the signatories are known to the receiving site. In the PC example, the PC chair might sign its agent and the PC member agents to indicate to sites that host agents that the task of these agents is to review papers.

From the point of view of infrastructure, the only thing that is assumed to exist between the sites of the agent system is the facility to share encryption keys. Little else is assumed since there is no standard key system in the security field. This is partly due to widespread debate over the most suitable key system for the internet: X.509 is regarded as being too complex and still lacking in real implementations; PGP is considered to have a certificate structure that relies on too much trust [39]. Simpler key system structures are now being developed, e.g., SDSI [26]. Such a key system is also being designed in the context of a CUI project [28]; there will be a collaboration of this work with JavaSeal in the context of a forthcoming SPP project that applies mobile agent technology in the area of domain name registration.

The issue being addressed by credentials and group-based policies is the assignment of trust levels, or access rights, to agents that arrive at a site. Consider the following example policies that a user might impose on his site:

- Only accept source or executable applets coming directly from JavaSoft, and that will not attempt to access the file system.
- Only accept agents from the Bank SBS to conduct transactions on site; agent must have visited SBS bank sites.
- An 'SBS bank agent' is one that the bank 'certifies' as being an 'employee'. Bank has rules on how it certifies agents, e.g., compiler must be known, developers must be contract company etc.
- Agent software contracted by SUN must not be made available on student (untrusted) machines.

A group calculation based on a set of trust rules and relations is initiated when an agent arrives at a site that is based on the credentials that the arriving agent furnishes. This process considers the agents origin and functionality, and will try to logically conclude a trust level that the agent can be assigned, where the trust level corresponds to the security privileges that the agent acquires. Here, the trust level is represented by a security group. A clear goal is clearly to formalise the decision making process into a logic of authentication and authorisation, in the style of [1], that caters for the distribution scenarios of internet systems. For example, if we were to say that sand-boxed JavaSoft agents are allowed visitor status (or be to be allocated the visitor group) on our site, we would have the rule:

```
sent-by(JavaSoft-grp), certified-by(JavaSoft-grp), sandboxed()
     becomes(Visitor-grp)
```

The rule is simply saying that any agent whose origin is the JavaSoft site, that bears a certificate to this effect and which satisfies the local sand-box criteria may *become* a member of the receiving seal's Visitor-Grp; this group is used by the seal in its subsequent authorisation decisions. The sent-by(), sand-boxed() and certified-by() actions would be implemented by the seal and operate on the credentials object; the *become()* action is implemented by the *chgrp()* method of the ACL policy.

A pioneering work in the field of authentication logic was [1] where a logic was developed for use to determine the access rights allocated to a process in a distributed system. Processes were part of a group and could be delegated a right for a file from another process, they could act in a certain role or group, and access was based on ACLs. This logic needs to be re-evaluated in the context of the internet where complicated and more dynamic trust relations exist, where the agent needs to authenticate the host, where there is no central authority and where several other methods of authentication are in use, e.g., proof-carrying code. Further, since we wish to integrate the group calculation process into the method security component, the algorithms used to derive the group of a component from its credentials using the rules must be tractable for an application's credentials and rules space, that is, there must be little overhead in calculating the group of an agent.

4.4 Police Agents

While the goal of the JavaSeal infrastructure is to provide security for mobile agents, it is also useful to turn this objective around, and to consider how agents themselves can be used to facilitate the enforcement of security.

Mobile agents bring several advantages to systems; these include 1) ease of software updates - new versions of applications are just sent out over the network to clients and installed dynamically, 2) active networks - instead of doing brute broadcasts of information, one can dynamically configure the route that each information packet travels by executing a decision agent at each site along the route, 3) support for asynchronous tasks (batch processing) - a user can send a program to work with a server while the user and his site is disconnected from the network, 4) scalability - a relatively simple application service interface suffices; new service functionality can be added within agents that are sent to the server and which process the results returned by the server, and 5) performance - emulating the execution of the client and server on the same site can reduce network traffic. These advantages can also be exploited by the security infrastructure; we refer to agents that fulfil some security role as *police agents*.

Concerning software distribution, agents aid the update of policies and policy components. For instance, a company with several sites typically uses the same security policy to control access to its information. These policies can change depending on who the company is currently doing business with, and updating policies is typically difficult to coordinate. Example policy updates include adding a new mandatory component to a policy environment, issuing a new password after a timeout, changing the ACL unit to include a new policy group, or changing the policy to remove the right of some user to access the information. This fits quite well into the JavaSeal model since policy components are classes or sub-seals that can be re-imported into the seal.

The active networking feature of agents is useful to help maintain trusted paths between sites and to reconfigure these paths when necessary. That is, each agent decides on the safest next site for the message and forwards the message to this site. This is very useful in secure systems since the degree of trust accorded to a site can vary, for example, with key freshness dating.

Examples of asynchronous security tasks - those that can be performed at the site to which the agent travels - include auditing and conflict mediation. The goal of an audit police agent is just to keep track of the resources that the main agent attempts to access. Conflict mediation is required when the policy of the visiting agent and the policy of host agents conflict. Recall that in the JavaSeal context, the environment policy component can be used for conflict resolution. In electronic commerce systems, trusted third-parties are often responsible for resolving conflicts and taking mediation actions [36]. This would be modelled in JavaSeal by sending the conflict mediator as a police agent, and importing it as a sub-seal into the environment policy component.

Service scalability is needed for security when new services need to be added to existing security servers. For instance, some finance applications require *separation of duty*, a requirement wherein each access to a document must be concurrently made by several clerks. This new functionality could be programmed as a police agent that interacts with a simple authorisation server. Another example is a police agent that adds an extra level of access control to ensure that the user who desires access to information is not on a local black-list. In the JavaSeal system, these police agents can also be represented as sub-seals used by the environment policy component of the environment seal.

Allowing police agents to travel a network poses questions of trust. Obviously, one cannot send a police agent to a site which one considers untrustworthy. In practice, this is not so restrictive. For instance, if a company trusts its server software to contact a security site to verify each access, then it can trust its server to verify access by using a police agent. In the same way, if a site is trusted to respect the decision of a conflict mediation, then it could be trusted to house the police agent that makes that decision. These examples also suggest how agents might address performance overheads brought by security message exchanges in a distributed system since remote service interaction is reduced.

Recall that not all seals can be, or are intended to be, moved. Seals are a structuring technique that are also used to represent services - authentication and key management services for instance are represented by seals. Like all services, they benefit from the secure environment implemented by their enclosing seals [33].

The examples of this section describe situations where it is intuitively advantageous to use agents in a security protocol. The problem is that these advantages are hard to quantify, except in terms of the messages exchanged, especially since trust is also involved. This is a problem with agent technology in general. Consider the case of a challenge response scheme for authentication, where a user forwards his name to the server, the server sends a challenge, which the user computes and returns the result. If the user uses an agent to compute the challenge, then it could calculate the response on the server's site. This means that the messages exchanged between server and user are never visible on the network and so the server cannot be subject to an interception and masquerade attack where an attacker intercepts a user's response and plays it himself. The visibility of messages is one measure for quantifying the advantages and disadvantages of agents.The Seal calculus is one area where such a study could be conducted [34] - visibility of messages could be modelled by the visibility of sub-seals, and this is already a basis for formally comparing computations.

4.5 Capsules & External References

Recall that seals are completely isolated in the sense that no object aliasing between seals can occur. Parameter passing between seals on method dispatch is by value. This means that each parameter object is copied into the destination seal. Further, all objects referenced by this object are recursively copied. A *capsule* is set of objects denoting the transitive closure of the original parameter object.

The problem with capsules is that they fail to preserve confidentiality - one cannot pass objects in capsules indiscriminately; some objects will contain sensitive information. In any case, the goal of security is not to blindly forbid access to information, but to control each access by intercepting each attempt to read the information and to dynamically decide if this access can go ahead. This assumes that a higher level authentication mechanism exists, e.g., credentials.

Value passing semantics also complicates sharing - ideally, a seal needs to pass out a name for an object that other seals wish to share. Whenever a seal wishes to access this object, it contacts the owning seal, furnishing the name as a parameter. In this way, the owning seal can make access decisions selectively for each object access request. One such naming approach integrated into JavaSeal is *external references*. Though the security and sharing issues could be overcome using an application-level naming scheme, this has the disadvantage that the JavaSeal kernel is unable to keep track of the propagation of names in the system.

4.5.1 External References

An external reference, or *Xref*, is an object reference for an object that may be located in a remote seal. As such, it seeks to model as closely as possible standard Java reference semantics. In particular:

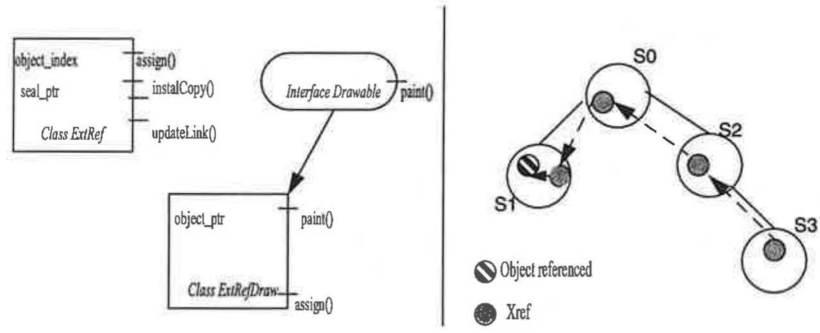
- There is security processing on use of the reference, (a seal can link its method security component to the reference)
- It enforces typing, that is, the Xref implements the same Java interface as the object referenced.
- An Xref is used for sending method calls, via the JavaSeal dispatch mechanism, to the named object.

An external reference is represented in a seal by a proxy object whose class is a sub-class of class ExtRef. This class contains a local flag (to indicate if the referenced object is contained in the local seal), a seal pointer to the parent or one of the children (which is followed on Xref de-reference if the object is not local) and an index that distinguishes the reference from external references for other objects. Each seal maintains a local mapping from an indices to external reference objects in order to locate an external reference in a seal.

To capture typing information, an object for which an external reference is being used must have a class that implements a Java interface. For example, an object of class ExtRefDrawable that implements the Drawable interface is an Xref for any object whose class implements the Drawable interface. This inherited class contains a reference that points to the referenced object if it is local and which otherwise is nil. The methods of an external reference forward a method call to the next external reference in the chain or to the object itself if it is local.

The class ExtRefDrawable contains a method *assign()*, that sets the external reference to point to a local variable of the same type. Thus for a declaration *v1:T; v2:XrefT*, the *assign()* method used as *v2.assign(v1)* models *v2=v1*; in contrast, the assignment *v1=v2* is illegal in this approach. We consider that *v2=v1* should be legal since an external reference designates an object in any seal, which does not preclude designated objects of the same seal as the Xref. On the other hand, the variable *v1* is declared to refer to a local variable, assigning it an object accessed via an Xref using *v1=v2* would entail security processing and mobility semantics on use of the reference that the reference holder does not expect.

An external reference for an object is first created in the seal where the referenced object resides. When an external reference is passed out in an inter-seal communication, the kernel detects that the parameter is a sub-class of ExtRef and so instantiates a new external reference in the called seal. The seal uses the index to detect future attempts to transfer the same reference into the seal; in this way there is only one external reference for an object per seal, and no cycles in an external reference graph can occur, unless through seals moving.



The left side of the figure shows the class structure for an Xref that references an object whose class implements the Drawable interface. The right hand side shows an Xref chain for an object in seal S1 - this object can be invoked using a series of method dispatches by seal S3.

Broken links can occur through mobility. The current approach is to detect broken links only when a call is made on an external reference. The seal itself must re-establish the link by setting a new seal pointer within the Xref, or by importing a new Xref through its parent.

4.5.2 Capsule Security

The capsule security policy component is responsible for verifying that no object is illegally transferred out of a seal in a parameter capsule during a method dispatch. This component can be designed to replace objects in a capsule that cannot be exported with external references for these objects.

A capsule is a graph G of objects; let nodes(G) denote the set of objects in G. The role of the capsule security component is to prohibit a specified set of objects, denoted F, from being a part of the capsule that is transferred out. Thus a capsule is secure if the sets nodes(G) and F are disjoint, and the capsule security component must be programmed to contain an engine that verifies this.

Since the JavaSeal system is not based on any changes to the JVM, each seal class requires a private method that returns the set of objects referenced within - and which is invoked by the capsule object recursively. Though awkward, the other solution would be to modify the Java serialisation code to include the capsule engine and which would verify whether the object is in F before serialising it.

What remains is to describe a way of defining a capsule security policy. Since F is a set of objects references, it must be a dynamically specified set. The user needs some way of giving a meaningful specification for F. The set F is represented within a security component class Capsule, and typically will contain entries for "special" objects such as terminals or any object that the seal specifies for F, via the *register*(Object o) method of capsule. The seal may also specify classes for F, that is, all instances of these classes automatically belong to F, in which case the parameter to the register method is a Java class object. Typical examples of classes that one would specify for F include password, or credentials.

4.6 Building Security Policies

Being able to implement an expressive range of policies is a key goal for JavaSeal. Yet expressiveness is a two-handed sword in a secure system since it can necessitate complex security mechanisms. The problem with complex security mechanisms is that users will not use them: if it is too difficult for a user to understand how to write a security policy, then he will not include any policy and so leave his site unprotected [5].

The environment a user possesses for programming his agents must also aid him to *design* and to *verify* his security policy. Designing a policy is about encoding it in the mechanisms of the system. In Unix, this means setting the ACLs; in an electronic commerce protocol, this means defining the message exchanges that distribute encryption keys among participants and the messages used to exchange money for services. Policy verification is about proving the policy design correct - that there are no back-door access in the system. Examples of back-door accesses include assigning an incorrect group to a user in Unix or failing to check whether an encryption key has expired in the electronic commerce protocol.

An important issue in policy design is thus to *make policy coding simple* for the application and agent developer. This can be done by using well-known protection techniques, such as Unix-like ACLs, or more generally by permitting users to reuse developed policy units when coding their own policy. Both approaches are supported in JavaSeal through a supply of policies (seal.lib package).

We mentioned in the introduction that a security infrastructure requires a formal model to reason about the policies in place. This is complicated in the agent context since an environment contains several policy components, for instance representing the mandatory policy or that of individual agents. One thus needs a components approach to designing and verifying security policies, that is, to specify a policy as a combination of individual policy components. One should be able to calculate the "value" of a composed security policy, and to compare this value with one's policy requirements, i.e., find out if the policy in place is secure enough. On the surface, this requirement is satisfied for the ACL access control policies presented. Here, the value of a policy is simply the set of groups that may access the seal. Further, since this defines a partial order on the set of all ACLs, we have a measure for comparing the security of one policy to another: a policy is stricter than another if it prohibits access by more groups. The partial ordering is also useful for defining a most secure upper bound for conflicting policy requirements: the most secure upper bound of two policies is simply the intersection of the two policies. Policy constructions and orderings have been studied in the context of authentication protocols [20] and access control [9][3].

Unfortunately, combining and comparing policies is not always so easy. Some security properties do not combine, that is, if two policy components enforce a secure property s, a policy combined may not necessarily satisfy s [22]. In the access control context, non-interference is one property that might not combine. A further problem mentioned earlier is that it might be intractable to calculate a value for a policy, i.e., find out what access states are possible. Again, we need to tailor the agent protection mechanisms used to be able to compose policies and so that some guarantees can be given about the security policy in place.

5 Discussion

JavaSeal [33] is a Java-based platform for building secure agent applications. The purpose of this position paper has been to outline areas for current and future work with respect to JavaSeal security. The key approach for security is to isolate agents or seals by reducing and fixing the set of variables shared between seals in the JavaSeal run-time, and then to reason about the degree to which these variables can be exploited as covert channels. The attention to covert channels and secure loading is one of the differences of JavaSeal to Aglets [18] or Mole [30].

Further, we are seeking an open approach in which we can model a range of mobile agent protection techniques. For instance, the Aglet security model in which sites or "contexts" specify access constraints which visiting agents must respect can be modelled by the environment security component of the JavaSeal security model; the same is true of [16][38]. For better protection of agents from hosts, voting schemes as used in the Delta-4 system [6] can be exploited to replicate agent execution and thus to isolate malicious sites by detecting and ignoring abnormal computation results. Alternatively, the data in the seal program can be scrambled [25][27][15]. Credentials can integrate any Java byte-code analysis algorithm [24][29][31].

Our requirements for a security infrastructure were then given. We currently use a simple framework based on ACLs, credential classes and capsules. Aspects that we wish to study in relation to this infrastructure include a suitable logic of authentication, the composition of security policies in seal environments as well as police agents. In particular, we insist on policies and mechanisms that are meaningful in an agent context: the policy of an environment can change dynamically, each agent has its own requirements to be fulfilled, and sites and agents must be protected from each other. Also important is that the framework be simple enough to allow automatic calculation of security, be this for assigning a group to an agent from its credentials or for calculating the access rights of an agent within an environment.

Apart from the ASAP project, the context for this work is also the forthcoming SPP sponsored DNX project and the current Esprit Coordina working group project. The goal of the DNX (Domain Name Exchange) project is to build a prototype system for registering domain names (www.name) using agents, based on the name service architecture furnished by CORE (the international Council of Registrars). The Coordina project is a working group studying the coordination paradigm. The relevance of this work to Coordina is that it deals with interacting autonomous agents that each have their own constraints that have to be mediated; the PC example cited is taken directly from a Coordina case study.

References

[1] M. Abadi, M. Burrows, B. Lampson, G. Plotkin, A Calculus for Access Control in Distributed Systems, in ACM Transactions on Programming Languages and Systems, 15 (3), June 1993, pp:1-29.

[2] J-P. Banâtre, C. Bryce, D. Le Métayer, Compile-time analysis for information flows. 3rd European Symposium on Research in Computer Security (ESORICS) 1994, LNCS 875, pp:55-73.

[3] C. Bryce, Security Engineering of Lattice-Based Policies, in 10th IEEE Computer Security Foundations Workshop, June 1997, pp:218-230.

[4] R. De Nicola, G. Ferrari, R. Pugliese, Coordinating Mobile Agents via Blackboards and Access Rights, Coordination Language and Models 1997, LNCS 1282, pp:220-237.

[5] D. Denning, Cryptography and Data Security, Addison-Wesley, Reading MA, 1982.

[6] Y. Deswarte, L. Blain, The Delta-4 Intrusion Tolerant System, in IEEE Symposium on Security and Privacy, May 1991.

[7] L. Dean, E. Felton, D. Wallach, Java Security: from HotJava to Netscape and Beyond, in 16th IEEE Symposium on Security and Privacy, May 1996, pp:190-200.

[8] W. Farmer, J. Guttman, V. Swarup, Security for Mobile Agents: Authentication and State Appraisal, 4th European Symposium on Research in Computer Security (ESORICS), Rome, Italy, September 1996, pp:118-130.

[9] S. Foley, Aggregation and Separation as Non-interference Properties, Journal of Computer Security, 1(3), 159-188, Januray 1993.

[10] L. Gong, Java Security: Present and Near Future, IEEE Micro, 17(3), May/June 1997, p:14-19.

[11] L. Gong, R. Schemers, Implementing Protection Domains in the Java Development Kit 1.2, in Proceedings of the Internet Society Symposium on Network and Distributed System Symposium, San Diego, California, March 1998.

[12] J. Gosling, B. Joy, G. Steele, The Java Language Specification, Sun Micro-Systems, ISBN 0-201-63451-1, August 1996.

[13] R. Gray, A Flexible and Secure Mobile Agent System, Proceedings of 4th Annual Tc/TK Workshop, July 1996.

[14] M. Harrison, W. Ruzzo, J. Ullman, Protection in Operating Systems, Communications of the ACM, 19(8), Augist 1976, pp:461-471.

[15] F. Hohl, An Approach to Solve the Problem of Malicious Hosts, University of Stuttgart Research Report, no. 1997/03, March 1997.

[16] T. Jaeger, N. Islam, R. Anand, A. Prakash, J. Liedtke, Flexible Control of Downloaded Executable Content, IBM Research Report, April 1997.

[17] T. Jensen, D Le Metayer, T. Thorn, Security and Dynamic Class Loading in Java: a formalisation, in International Conference on Computer Languages, ...

[18] D. Lange, M. Oshima, Mobile Agents with Java, in *Programming and Deploying Mobile Agents with Java*, editors Lange and Oshima, Addison-Wesley, to appear in 1997.

[19] J. Liedtke, Improving IPC by Kernel Design, Proceedings of 14th ACM Symposium on Operating System Principles (SOSP), Asheville, North Carolina, December 1993, pp:196-204.

[20] G. Lowe, Verifying Authentication Protocols with CSP, in 10th IEEE Computer Security Foundations Workshop, June 1997, pp:80-92.

[21] D. McCollough, A Hook-Up Theorem for Multi-level Security, IEEE Transactions on Software Engineering, 21 (4), June 1990, pp:563-568.

[22] J. Meyer, T. Downing, Java Virtual Machine, O'Reilly Associates, 1997, ISBN 1-56592-194-1.

[23] J-H. Morin, HyperNews: a Hyper-Media Electronic Newspaper Based on Agents, in Objects at Large, editor Dennis Tsichtitzis, University of Geneva, 1997.

[24] G. Necula, Proof-Carrying Code, in Proceedings of 24th ACM Symposium on Principles of Programming Languages (POPL), Paris, France, pp:106-119, January 1997.

[25] R. Rivest, L. Aldeman, M. Dertouzos, On Data Banks and Privacy Homomorphisms, in Foundations of Secure Compuations, ed. De Milo et al., Academic Press, 1978, pp:169-179.

[26] R. Rivest, B. Lampson, Simple Distributed Security Architecture, http://theory.lcs.mit.edu/~rivest/sdsi10.html.

[27] T. Sander, G. Tschudin, Towards Mobile Cryptography, University of Berkeley,TR-97-049.

[28] E. Solana, J. Haerms, Flexible Internet Secure Transactions Based On Collaborative Domains, TR in preparation.

[29] R. Stata, M. Abadi, A Type System for Java Bytecode Subroutines, Proceedings of Symposium on Principles of Programming Languages (POPL), January 1998.

[30] M. Strasser, J. Baumann, F. Holh, Mole - a Java-Based Mobile Agent System, 2nd ECOOP Workshop on Mobile Object Systems, Linz, July 1996.

[31] Sun Microsystems: The Java Language Specification, Release 1.0 Alpha 3, http://java.sun.com/doc/Overviews/java/index.html

[32] T. Thorn, Programming Languages for Mobility, in ACM Computing Surveys, 29 (3), September 1997, pp:213-239.

[33] J. Vitek, W. Binder, C. Bryce, Design Decisions of the JavaSeal Mobile Agent System, Also included in this report.

[34] J. Vitek, J. Castagna, Towards a Calculus of Mobile Computations, submitted to the IEEE Workshop on Internet Programming languages, December 1997.

[35] J. Vitek, M. Serrano, D. Thanos, Security and Communication in Mobile Object Systems, in Mobile Object Systems: towards the programmable inter-net, LNCS 1222, pp:177-199, editors: Vitek & Tschudin, 1997.

[36] M. Waidner, Development of a Secure Electronic Market-place for Europe, 4th European Symposium on Research in Computer Security (ESORICS), Rome, Italy, September 1996, pp:1-14.

[37] D. Wallach, D. Balfanz, D. Dean, E. Felton, Extensible Security Architectures for Java, in 16th Symposium on Operating System Principles (SOSP), Saint Malo, France, October 1997.

[38] J. White, Telescript technology: the foundation for the electronic marketplace, White Paper, General Magic Inc., 1994.

[39] P. Zimmermann, PGP Source Book and Internals, Boston MIT Press, 1995.