

Applying Security Policies Through Agent Roles: a JAAS Based Approach

Giacomo Cabri, Luca Ferrari, Letizia Leonardi

Dipartimento di Ingegneria dell'Informazione – Università di Modena e Reggio Emilia

Via Vignolese, 905 – 41100 Modena – ITALY

E-mail: {cabri.giacomo, ferrari.luca, leonardi.letizia}@unimo.it

Abstract. *Agents represent an emerging technology that grants programmers a new way to exploit distributed resources. Role is a powerful concept that can be used to model agent interactions, both between different agents and between agents and environments. Roles allow agents to dynamically acquire capabilities to perform specific tasks, and therefore enable separation of concerns and code reusability in software development and maintenance. Permissions and security issues related to the use of role should be carefully taken into account, especially when the agent scenario becomes open, including even mobile agents. In a Java agent scenario, we believe that the standard mechanism based on policy file does not suffice, because a fine grain permission management is required. This paper focuses on how to exploit the Java Authentication and Authorization Service (JAAS) at the role level in order to apply authorizations and local policies to Java agents for controlling the use of their roles.*

Keywords: *Roles, Java Agents, Authentication, Local Policies*

1 Introduction

Today's technologies for developing enterprise applications are commonly based on the traditional client-server paradigm, such as EJB [EJB]. This leads to a centralized architecture, which can hardly face issues arising in dynamic and variable environments. Emerging trends focus on collaborative distributed computing, and also on sharing information across the Internet. In this scenario, the use of *agents* can provide interesting solutions to face the dynamism and to grant the required adaptability. An agent is an autonomous software entity, which performs its tasks without requiring a continuous user involvement, being even able to play on behalf of its owner. The agent-oriented paradigm is emerging as a feasible approach for the development of today's complex software systems [Jen01, LucMP03]. In fact, the agent-oriented paradigm allows developers to naturally deal with distributed and concurrent environments, where different tasks are assigned to different agents, which can run simultaneously. This leads to the constitution of agent societies [Fas03] and, more in general, of multi-agent systems (shortly MASs), where agents compete and/or cooperate in order to complete their task(s). As a consequence, a key issue in the development of complex agent systems (such as MASs) is the management of interactions between agents, which must be designed and developed carefully.

The exploitation of roles represents a good paradigm to deal with agent interactions, thanks to the separation of concerns and code reusability it provides. The idea is to embed all interaction-related aspects (such as the communication protocols, commitments, etc.) into entities (the roles) that can be exploited by different agents. This grants a good modularity, allowing an agent to delegate some specific issues to the role it is playing, and thus keeping the agent code simple and easy to maintain. Furthermore, roles can be applied also to interactions between agents and other entities (such as databases), and the same roles can be applied to similar scenarios, thus allowing solution reusability. Of course, since roles embed the interaction logic, it is important to grant an appropriate security level for the actions an agent can perform through the roles it is playing. So far, several role approaches have been proposed

[CabFL04d], and, with regard to security in the role, one thing that seems common to all existing approaches, is that they conceive security at the whole role level, which means for example denying the role use if an agent is not authorized. Instead, in this paper we propose an approach with a granularity at the role operation level, thus finer than in other role approaches. To achieve this granularity, our approach, which is applied to Java agents and roles, relies on the use of the Java Authentication and Authorization System (JAAS) [JAAS, CabFL04e], thus resulting compliant to the Java 2 platform.

The paper is organized as follows: section 2 details some concepts about roles and security issues using them, showing also a simple application scenario; section 3 gives an overview of the JAAS architecture; section 4 provides an overview of our approach while section 5 gives a more complex case study, completed by the related Java code; section 6 reports related work and compares our approach to similar ones. Finally, section 7 reports conclusions.

2 Roles at work

The role concept represents a powerful design pattern to model agent interactions, allowing separation of concern and code reusability in software development and maintenance [CabFL03, BecGKM99]. Any role can be thought as a stereotype of a behaviour common to different agents, and, in a more practical way, it can be defined as a set of capabilities and an expected behaviour [CabFL03] that agents can exploit during their life. Exploiting a role means that the agent is in charge of assuming it, using then the capabilities that the role itself grants, and releasing it when it is no more needed. The dynamism of the assumption/release process is in charge of the role system implementation. On the one hand, this process can be static, meaning that roles are assigned to agents at the design phase. On the other hand, it can be very dynamic, meaning that an agent can choose the roles it is going to play at runtime, even on the base of gathered information from the execution environments. Roles allow the reuse of both code (i.e., the role implementation) and experience (i.e., the role purpose), allowing developers to build role libraries and making possible for other developers to use these libraries for similar purposes in different scenarios. This leads to a design/development process that is similar to the component paradigm [EJB].

Another interesting characteristic of roles is that they can be exploited to manage permissions, allowing more secure execution environments and, in fact, they have already been exploited in security control systems, such as the Role Based Access Control [San96]. Since roles are mainly used to manage interactions between agents, it is possible to embed into roles security constraints, in order to get control over the actions performed through the roles themselves. To better explain this concept, consider a user agent (UA) in charge of voting on behalf of its owner in an e-democracy scenario [CabFL04c]. In this situation, it is possible to develop a voter role that a UA can exploit to register the user's vote in the e-democracy database (see Figure 1). Since a UA, acting on the behalf of its user, exploits the role capabilities, it is not in charge of knowing the details concerning the vote registration: the *role* does "know" how to connect to the e-democracy system and to put the vote. In order to take into account security, the role can be developed to exchange only encrypted data with the e-democracy system, which means the vote will be encrypted before it is sent to the database. This simple example shows how it is possible to exploit roles in order to enforce security and to take control over interactions. Furthermore, the above example represents also a case of use of roles that enforces local policies; in fact, if the database does not accept connections from UAs that are not playing the voter role, each agent is forced to use that role (and thus to use the cryptographic algorithm embedded into the role itself) to put its vote. Moreover,

since roles are tied to the local execution context [CabFL03], each different e-democracy scenario is free to implement or not cryptography, as to choose the algorithm to adopt.

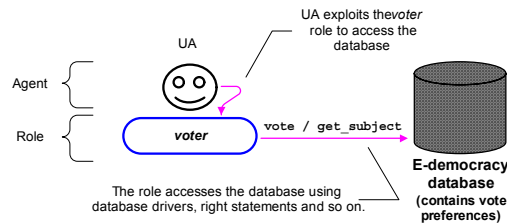


Figure 1 - An agent exploits the voter role to express the user's vote.

Several role approaches have been proposed [CabFL04d], each one with its own concept of security through roles. In general, the role approaches adopt a point of view about security that considers any role as a whole, which means they base security by allowing/denying the assumption of a certain role. It is important to note that security cannot be applied only from a static point of view, but must take into account run-time conditions. This means that a security policy must be as dynamic as possible and therefore the role system itself must be dynamic. Nevertheless, a security mechanism based on the whole role implies that roles must be developed as small entities, providing as few as possible capabilities, otherwise denying a role assumption will produce the denying of safe capabilities. Let us consider again the example of Figure 1, and suppose that the voter role provides two capabilities: *get_subject*, which checks for the current voting subject, and *vote* that performs the voting action. Now, imagine that a UA does not own the rights to vote (e.g., it represents an under age user), and imagine that, as the most of role approaches do, it is not allowed to assume the voter role. In this way, such agent will not be able to exploit any capability provided by the voter role, even the safe *get_subject*. This leads to an awkward situation, since in a role system that performs a security policy at the whole role level, the only way to permit the under age UA to exploit the *get_subject* service is to embed the above capability in a separated role, allowing the UA to assume the one with the safe service and denying the assumption of the voter role. This will lead to the production of a lot of roles, since the capabilities embedded in anyone of them will reduce in order to allow a fine grain in security control. As a consequence, the effort of role developers increases and the assumption process results complicated, since now involves several roles [CabFL05].

Starting from the above considerations, we decided to propose an approach that enforces security on role capabilities, in order to perform security checks with a fine grain, focusing on each role operation. Our approach has been integrated in the RoleX environment [CabFL05], producing a second version of this Java role system for mobile agents with a high degree of dynamism in the role assumption/release process. In fact, the first version of RoleX only provided support for Java permissions associated to roles. Even if the approach described in this paper has been applied to RoleX, it can be integrated into several role systems, because it is very general.

3 Introduction to JAAS

The Java Authentication and Authorization System (JAAS) [SUNJA], provided with the Java 2 platform, is a mechanism used to manage user authentication in Java programs. All main classes are contained in the *javax.security.auth* package, but programs that use JAAS need to interact with only three classes of them: *LoginContext*, *Subject* and *PrivilegedAction* (see Figure 2).

In JAAS, authentication is performed through a login action, which, if successfully done, provides also a set of extended permissions and, thus, authorizations. Logins are related to a specific *context*, that represents an application scenario. The use of contexts allows the definition of a modularized system, where each single scenario can exploit its own login procedures. Moreover, thanks to the use of login contexts, the same component (or user) is able to login into different scenarios at the same time, exploiting thus the acquired permissions for each context; on the other hand, the use of contexts allows administrator to define permissions associated to each action with a fine granularity.

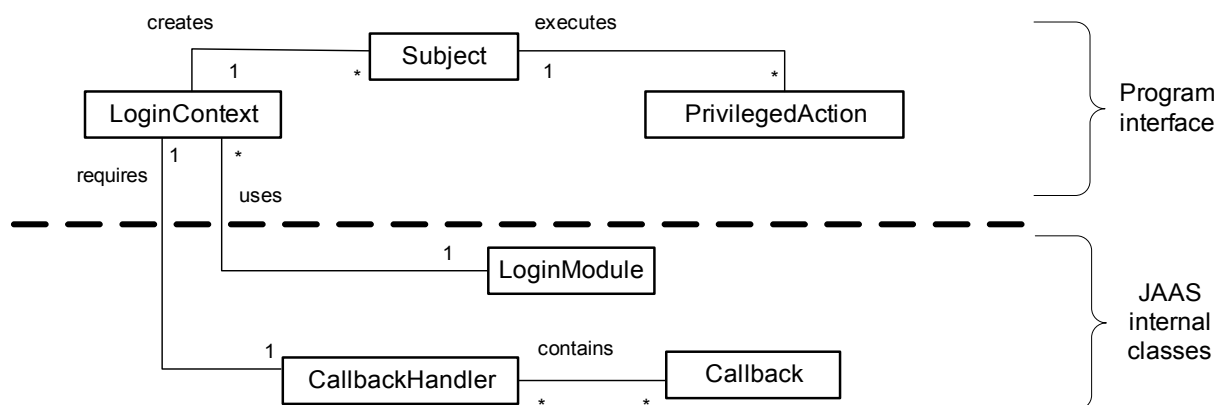


Figure 2 - JAAS main component classes.

The first JAAS component an application must deal with is the *LoginContext*, a class in charge of defining the JAAS structures for a login into a specific context. In fact, the application requires to the *LoginContext* the login for a specific context, then the *LoginContext* creates all JAAS objects required to manage the login for that context (see Figure 2). In particular, the *LoginContext* reads a configuration file, called *JAAS Configuration File*, that specifies, for each context, which *LoginModule* must be used, and further, the kind of login. The *LoginModule* is a component in charge of checking the login data (e.g., a password) in order to establish if the login can be successfully done or not. Through the JAAS configuration file, administrators can change the login procedure, defining that a specific login context must be handled by a pair username/password, by a shared secret (e.g., a PGP key), etc. The *LoginModule* is the implementation of the login procedure; for example, for a login driven by a couple username-password on a Unix system, the *LoginModule* could be a reader for the */etc/passwd* and */etc/shadow* files. Login modules are stackable, and a JAAS developer (or an administrator) can specify that a login procedure must be performed through several login modules.

If the login succeeds, (i.e., the *LoginModule* has verified that login data is right), the *LoginContext* returns an instance of *Subject*. A *Subject* object represents a certificate for the login, and means that the entity that required the login is now authenticated in the context. Please note that, even if an entity has already authenticated itself in JAAS, every operation must be done presenting the *Subject* object as a login certificate, otherwise the operations will be performed as unauthenticated entity. This also allows an entity to perform certain operations with different certificates (i.e., logins) depending on the permissions required to perform such as operations.

Once authenticated, operations must be performed through the *doAs(..)* method of the class *Subject*, which presents the following prototype:

```
static Object doAs(Subject cert, PrivilegedAction op)
```

where *cert* is the *Subject* instance obtained at the login time, *op* is the wrapper for the operation to perform (detailed later in this section) and the return value (of the operation) is a generic *Object*. Please note that the above method is static, that means that the *Subject* class has a double meaning: when used as a object instances they represent successful login certificates, when used as a class it represents the way to perform authenticated operations with permissions acquired at the login time. The use of the *doAs(..)* method is very similar to the use of the *doPrivileged(..)* method of the class *AccessController* of the Java 2 platform, the only difference is that in this case the method requires the login certificate while the latter does not.

A *PrivilegedAction* represents a wrapper around the real operation to execute, thus the JAAS system can execute the operation through a common API interface. In particular, it is the *run()* method of the *PrivilegedAction* class that wraps the operation to execute. In other words, to obtain a wrapper of the operation, it is required to write a class implementing the *PrivilegedAction* interface, and to put the code to execute into its *run()* method, thus JAAS will be able to execute such as code as a privileged operation.

In Figure 2 there are other components of the JAAS architecture not explained yet. Two strictly related one each other are *Callback* and *CallbackHandler*. Callbacks are modules in charge of storing login data (e.g., a password) in order to allow their further evaluation from the *LoginModule*. A *CallbackHandler* is a container of callbacks, which eases the use of a set of callbacks (even if the number of callbacks is not known a priori). What happens is that the *LoginContext* passes the *CallbackHandler* (and thus all *Callbacks* contained in it) to the *LoginModule* that can evaluate login data.

The last concept that must be explained is about (extended) permissions, granted by a successful login. In a standard Java application, permissions are managed by the *SecurityManager* and the *AccessController*, and they are defined and stored in a policy file (usually the *.java.policy* file). In JAAS, another policy file is added¹, and this file contains the definition of permissions that must be granted to an authenticated entity once it has logged in.

Please note that, the authentication phase is in charge of the application itself, which means that the application should ask “manually” the JAAS system to authenticate itself. In other words, the JAAS system does not ask explicitly the application to authenticate itself, but simply denies some operations if the application has not authenticated itself (i.e., if the application does not provide a valid *Subject* instance).

4 Implementing a JAAS Based Role System

In the previous section, readers have been introduced to the concept of role and to the JAAS architecture. This section shows how our approach merges both ones in order to obtain a JAAS-based role implementation.

4.1 Motivating the use of JAAS

Our approach provides two kinds of fine grain permission managements [CabFL04e]: it allows to take control over (i) any single operation and over (ii) any specific agent instance, depending on the run-time constraints. The first type of granularity has already been introduced with the example of Figure 1; the idea is to authenticate the couple agent/role operation before the agent can perform a role operation. In this way, if the agent is successfully authenticated, it can continue performing the role operation,

¹ There is not a specific name or location for this policy file.

otherwise it cannot. The authentication is not meant at the role level, in the sense that each possible dangerous role operation must perform an authentication before it proceeds. The second kind of granularity can be obtained with a pre-authentication process, which will be detailed in the following.

The Java standard SecurityManager, based on a policy file mechanism, is not enough for our purposes since it allows only to set permissions for groups of entities, such as agents (e.g., “all agents coming from host XYZ cannot write in the local filesystem”), without control over single agent instances because the authentication is done at the agent class level. To have a fine grain permission management we propose to apply JAAS to Java agent contexts.

Agent applications are not like standard applications, because of their dynamism and since they are less user-interactive. Therefore, the JAAS architecture does not suffice as it is, but there is the need to build a more specific authentication mechanism, able to autonomously decide at runtime if an agent can be authenticated. This can be done by means of a *supervisor agent* (SA), launched by each host administrator, which can monitor the system extending the JAAS authentication decision system.

Application agents have to ask the SA to authenticate them, before performing role operations, by means of JAAS callbacks. In fact, the SA gives the right callbacks to the agents that request authentication, so that they can interact with JAAS to obtain an effective authentication. Nevertheless, the fact that an application agent obtains the callbacks does not mean it will automatically be authenticated by the JAAS system. Therefore, using a SA the authorization process is split into two parts (i.e., pre-authentication and authentication). The SA grants a pre-authentication, based on runtime information, while the JAAS architecture, managed by the human administrator, provides the latter authentication step, statically set.

4.2 Pre-Authentication

In our approach, the SA assumes a role (authenticator role) to authenticate other agents/roles. Since roles tied to the same application context are developed the one with regard to the other [CabFL03], they can know which is the role that the SA will assume to authenticate them, becoming the authenticating entity. Starting from the above consideration, each role in a context is able to find the authenticator entity (i.e., the supervisor agent that is playing the authenticator role for that context), communicating with it to get callbacks. As shown in Figure 3, the SA can assume the right authentication role for each of different application contexts to authenticate requests coming from different contexts.

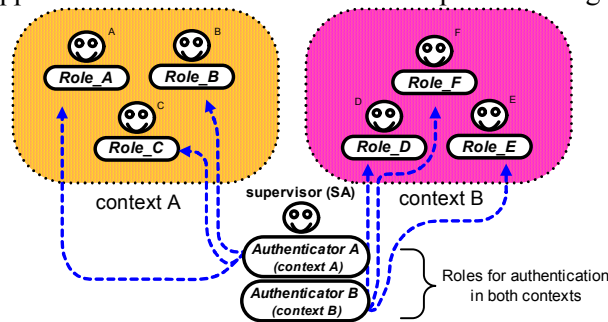


Figure 3 - The supervisor agent can exploit different roles to manage different contexts.

4.3 Authentication

As shown in Figure 4, for example, the agent A plays the role Role_A belonging to the context A, needs to be authenticated and for this reason (a) asks its context-specific authenticator role Authenticator A (owned by the SA) for the callbacks. Thanks to its role, the SA sends back the needed callbacks (b) and

then (c) the agent A and its role Role_A use these callbacks to authenticate themselves in the JAAS architecture.

If the JAAS authentication is successful, as detailed in section 3, an instance of the Subject class will be returned from JAAS; this instance represents an operation certificate, needed to continue with the role operation execution. In fact, our approach reaches the role operation granularity by using the above operation certificate as a requested data for the operation execution: if the agent does not own the operation certificate, it cannot execute the role operations. Furthermore, each dangerous role operation should require a different operation certificate from the other operations, thus the agent that has obtained a certificate for one operation cannot exploit it to perform another operation.

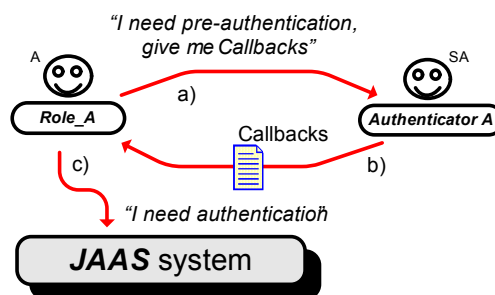


Figure 4 - Requesting callbacks to the supervisor agent, and then authenticating into JAAS.

4.4 Acquiring Agent Data for JAAS

Since a mobile agent can move from one host to another, it must carry authentication data from the home node (from which it is launched) during its trip path. User's authentication is simpler than agent's one, since callbacks can acquire the needed data by asking the user in an interactive way. In the case of agents, this cannot happen, so that callbacks must extract authentication data from agents.

First of all note that an agent should carry data in a suitable way for roles, so that agents can provide them to callbacks. A good way to transport "personal" data, so that it can be used by roles, is the exploitation of hash-maps [CabFL03]. With this approach a callback that, for example, needs to acquire the "home node", can simply search for it in the map.

Please note that application agents do not know exactly which data (i.e., which map entries) are required to authenticate them, since each target host can require different data. Nevertheless, since roles are tied to any local context, roles related to the same context are developed the one with regard to the others, each authenticator role assumed by the SA knows how application agent roles work, and which data can extract from agents. This means that the authorizer role returns callbacks developed according to the other agent roles, so that required data can be easily extracted.

5 A Case Study

This section presents a concrete mobile agent application, based on roles, that has been treated as a case study for the proposed JAAS authentication mechanism.

5.1 The MailConfigurator Application

The application, MailConfigurator [CabFL03b, CabFL04], aims at helping users registering e-mail accounts and then configuring, transparently, their e-mail client programs.

The MailConfigurator behaviour can be summarized in the following steps (see Figure 5):

- a) the user who wants to register a new account is in charge of starting the application on her host, that means a new mobile agent (called *personal agent*) is created. The personal agent (PA) collects preferences about the e-mail account to subscribe (e.g., protocol type, minimum available space, etc.), and then starts a trip around different mail providers in order to find the one that can match the user's wills;
- b) on each mail provider, the PA interacts with a resident agent, called *mail provider agent*, that is in charge of administrating the mail provider software, accepting, processing and allowing/denying the incoming requests from the PAs;
- c) in the case the interaction between the above kind of agents is successful (i.e., the mail provider has registered the requested account), the mail provider agent returns a *registration voucher* to the PA; the voucher contains all data required to configure the user e-mail client (e.g., host name, port, protocol, etc.). If the interaction is unsuccessful, the personal agent moves to another mail provider and repeat the interaction with its mail provider agent;
- d) once the PA has obtained the registration voucher, it comes back home and configures transparently the user's e-mail client, notifying she when done, and thus she can start immediately using her new e-mail account.

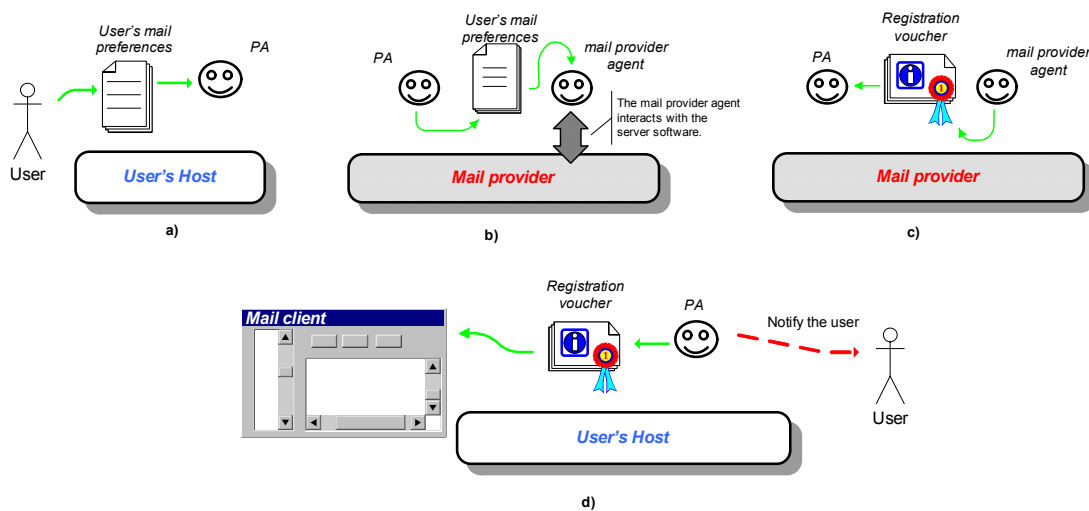


Figure 5 - Steps of the MailConfigurator application.

As shown in Figure 5, the mail provider agent interacts with the mail provider software in order to check if the account registration can be done or not. For example, the mail provider agent has to check if the requested username is already in use or not, or if the requested protocol is supported, and to do so, it must “ask” the mail server software. In other words, the mail provider agent acts also as an interface to the mail provider software, allowing the PAs to indirectly interact with the latter. The use of an intermediate agent (the mail provider agent) as interface to the mail server software grants security, scalability and portability. Security because the mail provider agent can apply security constraints during the interaction with the PAs, in order to verify their credentials; scalability because the mail provider agent can be cloned, leading to a MAS where each mail provider agent is in charge of serving a specified number of requests, and finally portability since the mail provider agent can exhibit the same interface independently of the mail provider software that it is “masquerading”.

5.2 Exploiting Roles in MailConfigurator

In the above subsection, the MailConfigurator application has been described only talking about agents, but it deeply exploits roles in all the phases and for all agents. In fact, the PA exploits three different roles depending on the situation it is living in, while the mail provider agent exploits a single role (see Table 1). The exploitation of roles makes the application adaptable and flexible, granting also a good degree of maintainability. Thanks to roles, the development of the application components (i.e., the agents) is simple and fast, since roles allow to clearly uncouple concerns. For example, considering the first phase of the application, when the personal agent collects the user's preferences, it is possible to produce different implementations of the `data_collector` role so to meet different user behaviours: there can be an implementation that interactively asks the user her preferences through a set of dialog windows, another that performs the collection through a text based interface, or using a speech recognition engine, or in a non interactive way, etc. The point here is that it is possible to quickly change the behaviour of the personal agent, changing its role implementation, while the agent and its mobility logic remain the same. In other words, thanks to the use of roles, it is possible to divide the development of the agent into two main parts: (i) the agent and its search logic and (ii) the interaction logic.

Similar considerations can be done for all the other roles of the personal agent. For example, different implementations of the `parameter_setter` role can handle different e-mail clients, allowing the personal agent to configure each specific client. The subscriber role, instead, is used by the personal agent in order to provide to the mail provider agent information (i.e., user's preferences) in a way it can understand. For example, some mail providers could require a cryptographic exchange of data between agents, due to security reasons, thus the subscriber role is in charge of knowing the cryptographic algorithm used on that host. Here, again, it is possible to note how roles ease the development: if a provider decides to change its cryptographic algorithm, it is in charge only of providing a new subscriber implementation, while the personal agent will not require any change. Furthermore, since roles are dynamically exploited at run-time and are locally tied (i.e., they belong to a context and are not moved with the agent), an agent is free to move across different hosts exploiting different role implementations on each of them. Thus, for example, a personal agent could use a cryptographic transmission on a mail provider, and a plain data exchange on another, and this happens transparently.

<u>Role name</u>	<u>Exploited by</u>	<u>Exploited to</u>
<code>data_collector</code>	<i>personal agent</i>	Acquire the user's preferences for the e-mail account the agent has to register.
<code>subscriber</code>		Interact with the mail provider agent.
<code>parameter_setter</code>		Configure the user's e-mail client.
<code>administrator</code>	<i>mail provider agent</i>	Interact with the personal agent.

Table 1. Roles in the MailConfigurator application.

The last role shown in Table 1, `administrator`, is exploited by the mail provider agent. This role has a double meaning: (i) it allows the agent to rightly interact with the mail server software and (ii) allows the agent to understand and use the data provided by the personal agent.

Due to the above explained roles, the Figure 5 changes into Figure 6 showing each exploited role; please note that all the above agent interactions happen thanks to and through roles.

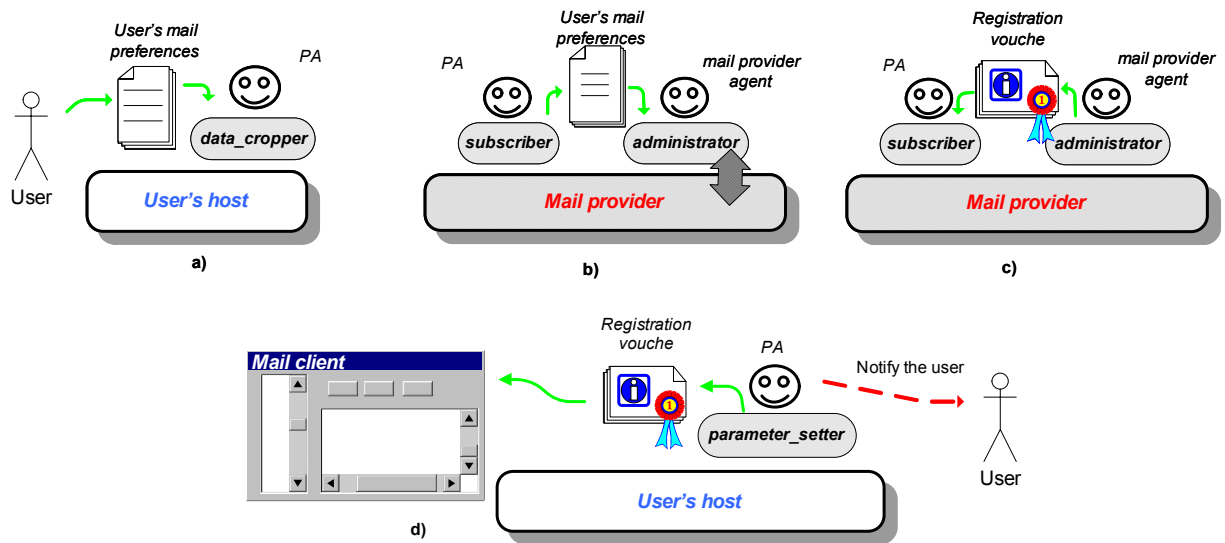


Figure 6 - Steps of the application redesigned with roles.

5.3 Security Problems in MailConfigurator

The application described above can suffer from different security risks, in particular due to the fact that the personal agents are in charge of transporting the user's preferences and, in case of success, data that can be used to access the user's e-mail account. Moreover, it is possible that faking and malicious personal agents try to do actions they cannot. For example, a malicious agent could try to directly interact with the mail provider software, or to wrongly configure an e-mail client. While the protection of sensible data (e.g., the user's preferences) is not the subject of this paper, the latter cases are those this paper focuses on: the control of agent actions, being these actions done through a particular role.

Embedding a security mechanism into the role the agent is going to assume, allows the system to recognize a malicious behaviour just before it happens. This is an active way of enforcing security, since it operates at a fine grain and at run-time, depending on the dynamic agent behaviour.

5.4 Code Samples

This section shows some code fragments related to the use of JAAS into roles, taking into account the application MailConfigurator. In particular, with regard to the role `parameter_setter`, this section shows how to protect the execution of a possibly dangerous task, like the access to the configuration of the user mail client. The specific role operation is called `configureClient`, and is implemented through the namesake method in the above role.

In order to focus on JAAS, this section will not show the complete implementation of the selected role, using the code as a pure sample. Please note that, due to the JAAS architecture (see section 3), the role developer is in charge of defining several classes other than of the role itself, and in particular a login module and one (or more) callback are required.

5.5 The Operation Wrapper

The first step is to define the operation wrapper for the `configureClient` operation, and to embed it into a *PrivilegedAction*. This section does not cover the details about the configuration of the e-mail client, since those details are too strictly related to the mail client type; nevertheless please note that it suffices to change the *ClientConfigurator* class to provide a different e-mail client support.

To define the operation wrapper, the programmer is in charge of building a class that implements the *PrivilegedAction* Java interface and that performs the operation code in the *run()* method, as in Figure 7.

```
public class ClientConfigurationOperation implements PrivilegedAction{
    // the mailbox registration parameters
    MailboxParameters par;

    // constructor
    public ClientConfigurationOperation(MailboxParameters params){
        par = params;
    }

    // operation code
    public Object run(){
        // get the configurator for the kind of client
        // (automatically from a system property)
        try{
            ClientConfigurator conf =
                MailConfiguratorFactory.getClientConfigurator();
            conf.writeConfiguration(par);
        }catch(UnsupportedClientException e){}
    }
}
```

Figure 7 - The e-mail client configuration code.

As readers can see, the code of Figure 7 stores the configuration parameters of the registered mailbox (e.g., outgoing server name, port, etc.) and then uses that parameters to perform the configuration in the *run()* method. Please note that the latter is executed every time the agent calls *doAs(..)* of the *Subject* class (see section 3).

5.6 The Role Code

Once the operation has been written, the programmer can embed it in the role code. Since the configuration operation is represented by the *configureClient()* method of the role, the code of Figure 8 could be a possible implementation of such as behaviour.

The first step is to obtain the callbacks that must be used for the JAAS login, and as detailed in section 4.2, this is done through a pre-authentication. Pre-authentication implies to contact the SA and to obtain, through its authorizer role, the set of callbacks; the related code is not shown in the figure, since it is tied to the role system implementation. As an example, in RoleX, the pre-authentication could be done sending a request event to the supervisor agent, which replies with the given callbacks.

The bold part of Figure 8 performs the login into the JAAS architecture. As detailed in section 3, the first step is the use of a *LoginContext* object, that is built specifying the callbacks and the context identifier (in this case *parameter_setter_role*). Then the login is tried, calling the *login()* method; in case of success, an operation certificate, as *Subject* object, is obtained by the *LoginContext*. Finally, using the static method *doAs(..)*, an operation of kind *ClientConfigurationOperation* is executed. As already stressed, the difference between the execution of such operation through the *doAs(..)* method or not, affects the permissions with which a role operation is executed, which depend on the operation certificate (i.e., the *Subject* instance) the role has obtained from the login.

An important thing to note here is that the PA has no view of the JAAS login, which is dealt with into the *parameter_setter* role, thus the agent (and its developer) has not to worry about JAAS

details. Of course, it is important that the role has access to the agent credentials, since they will be used for the login; however the credentials can be easily passed from the agent to the role (and viceversa) by means of a hashmap [CabFL03].

```
public class parameter_setter{
    // the parameter acquired from the registration process
    private MailboxParameters pars;

    // credentials of the agent
    Hashmap credentials;

    void configureClient() {
        // acquire callbacks from the supervisor agent
        // (pre-authentication)
        Callback cb[] = . . .
        // create a callback handler
        Mail_callbackhandler cbh = new Mail_callbackhandler (cb,credentials);
        // now login into JAAS
        try {
            LoginContext lc = new LoginContext("parameter_setter_role" , cbh);
            if( lc.login() ) { // execute the operation
                Subject operationCertificate = lc.getSubject();
                Subject.doAs(operationCertificate,
                            new ClientConfigurationOperation(pars));
                lc.logout();
            }
        }
        catch( LoginException e ) { // login failed }
    }
}
```

Figure 8 - The code of the role operation.

5.7 Callback and CallbackHandler Implementations

As detailed in section 3, the *LoginContext* and the *LoginModule* use a set of *Callbacks* in order to acquire the login data, for a further evaluation. Starting from this consideration, there must be an implementation of each callback, such as of a *CallbackHandler*, which is a container for a set of related callback objects.

The definition of a callback can be quite straightforward, since the only thing to do is the implementation of the tagged interface *Callback*, as shown in Figure 9. The shown callback simply acquires, from the agent credentials, the previous host the agent has visited; the value is then stored in a protected field (*previousHost*), for a further evaluation that will be done calling the *getHost()* method.

```
public class PreviousHostCallback implements
javax.security.auth.callback.Callback {
    protected URL previousHost=null;
    public void acquireData(Hashmap credentials) {
        // search for the previous host in the credentials hash and store it
        this.previousHost= credentials.get("previousHost");
    }
    public String getHost(){ return this.previousHost.toString(); }
}
```

Figure 9 - An example of Callback.

Please remember that developers can provide as much callbacks as they need, even more complex than the one shown in Figure 9. Once callbacks are ready, they must be put into a handler, thus developers must provide an implementation of a *CallbackHandler*, like the one shown in Figure 10.

```
public class Mail_callbackhandler
implements javax.security.auth.callback.CallbackHandler {

    // keep the agent credentials map
    private HashMap credentials = null;

    // all my callbacks
    Callback cb[] = null;

    public Mail_callbackhandler (Callback calls[], HashMap hm ) {
        this.cb = calls;
        this.credentials = hm;
        this.handle( this.cb );
    }

    // set callbacks data
    public void handle ( Callback []cb ) throws
    IOException, UnsupportedCallbackException {
        for ( int i=0; i< cb.length; i++ ) { // cast the callback and set its data
            if ( cb[i] instanceof PreviousHostCallback ) {
                ((PreviousHostCallback)cb[i]).acquireData(this.credentials );
            }
            else
                if ( cb[i] instanceof ... ) {...}
            else
                throw new UnsupportedCallbackException();
        }
    }
}
```

Figure 10 - The callback handler implementation.

As readers can see, the callback handler in Figure 10 simply stores the set of callbacks (as an array) and the agent credentials, in order to pass them to the callbacks that require to know something about the agent. When the *LoginModule* or the *LoginContext* need to deal with callbacks, the method *handle(..)* of the interface *CallbackHandler* will be called. As shown in the figure, the method iterates on each callback, casting it to the right type and requiring the callback to acquire data. Before this moment, no data is acquired by any callback, even if the handler has already all the credentials of the agent.

Since the callback handler has been exploited, its method *handle(..)* has been called to collect all the data required for the login. In other words, the *handle(..)* method must initialize in the appropriate way each callback, thus the login will proceed with the required data.

5.8 Login Modules: Validating Agent's Login

Callbacks and callback handlers are used only to acquire agent's credentials, waiting for their later evaluation that is done through a *LoginModule*, which is in charge of analyzing agent's data at runtime, allowing or denying the login.

As detailed in section 3, the *LoginModule* depends on the application context, and the binding between a particular module and its context is set in the JAAS configuration file. This file must contain

the fully qualified name of the class used as login module, in order to instrument the *LoginContext* about the instance to create and use as *LoginModule* once the request of login in a context arrives. As shown in Figure 11, for the context `parameter_setter_role` the class `brain.MailConfigurator.MailLoginModule` must be created and used as login module. Please note that the context is the same specified to the *LoginContext* as in Figure 8. The keyword *REQUIRED* means that the login can be considered as successfully if and only if the *MailLoginModule* confirms it.

```
// in the configuration file
parameter_setter_role { brain.MailConfigurator.MailLoginModule REQUIRED; }
```

Figure 11 - The JAAS configuration file entry.

With regard to the code of the login module, a possible implementation could be the one reported in Figure 12.

```
public class MailLoginModule
implements javax.security.auth.spi.LoginModule {
    private CallbackHandler cbh = null;
    private Principal agentPrincipal = null;
    private Subject mySubject = null; // get from the initialize() method
                                     // automatically

    // login method
    public boolean login()
    throws LoginException {
        try {
            success = true;
            Callback[] cb = this.cbh.cb;
            for( int i=0; i<cb.length; i++ ) {
                if(cb[i] instanceof PreviousHostCallback &&
                    && !((PreviousHostCallback)cb[i]).getHost.equals("mailprovider.edu")){
                    success = false; agentPrincipal = new ...;
                }
            }
            else // other checks
                return success;
        } catch( Exception e ) {
            throw new LoginException();
        }
        //complete the login
        boolean commit() throws LoginException{
            mySubject.getPrincipals.add(agentPrincipal); return true;
        }
    }
}
```

Figure 12 - The LoginModule implementation.

The most important method of each login module is the *login()* one, that is called through the *LoginContext.login()* one. As shown in Figure 12, the method is in charge of evaluating data acquired from callbacks through the callback handler, and thus the method extracts each callback contained in the handler and tries to cast it to the right type, for evaluating the data. As an example, this code shows the extraction of the previous host from the *PreviousHostCallback* of Figure 9, checking if the value is equal to the host `mailprovider.edu`, that is supposed to be the only mail provider the user has confidence in². If the last host does not correspond, the login is denied, otherwise another callback is checked.

² The test on the name of the last visited host is shown only to keep the example simple, and to allow readers to focus on concepts strictly related to the JAAS proposed implementation. Production code should perform more complex tests.

Supposing that all callbacks provide data evaluated as right for the login, the login can be successfully done, and since the above module is the only one required to complete the login, the agent can be authenticated for the specified context. The role operation can now proceed, thus the agent will execute the role operation with different privileges than without using JAAS.

6 Related Work

This section shows a few approaches related to security applied to agents and roles, briefly comparing them to our approach.

The Naplet system [XuF03] is a mobile agent system that exploits some Java security concepts, like *subject* and *login context*, in order to implement a secure environment for the execution of agent services and actions. In this system, each agent is a naplet, that is an agent associated with a set of principals or credentials. Each time a naplet (i.e., a mobile agent) arrives to a remote host, the host allows the execution of the agent code through the Java Subject object obtained from the principals/credentials the naplet carried on. In other words, from the naplet principals/credentials, the remote host extracts a Subject object, that is then used to invoke methods (i.e., execute the naplet code) on the naplet agent itself, as shown below:

```
...
Subject napletExtractedSubject = Runtime.authenticate(naplet);
                                   // the "naplet" is the agent object
AccessController.checkPermission();
Subject.doAsPrivileged(
    napletExtractedSubject, new PrivilegedAction(){ naplet.init(); }, null);
...
```

The extraction of the Subject object from the naplet, and the following code execution is performed by the naplet server, an entity in charge of receiving and authenticating the incoming agents; in other words, the Naplet architecture cannot work without the support of an appropriate naplet server.

The Naplet approach is really interesting, even if it has a few drawbacks when compared to ours. First of all, while our approach can be easily integrated into existing agent-based systems, Naplet requires the use of specific agent classes, as a specific server as well. This means that, instead of integrating a security approach into existing mobile agent platforms, Naplet proposes a completely different mobile agent approach, which takes care of security issues. A second drawback of Naplet is that it is focused on a whole agent (a naplet) approach instead of on a single agent service approach. Even if, as Naplet authors state, the system can be used to implement a role based secure architecture, an access control on each single service of the agent could become awkward.

Another interesting approach, specially role-oriented, is GAIA [WooJK00], that defines a methodology for the definitions of organizations of agents, where different roles interact. From a security point of view, GAIA associates to each role a set of permissions, which are rights that are further associated to the agent playing that role. GAIA does not define a formal notation to define roles, but exploits a formal notation to express permissions associated to roles, and thanks to this notation designers and developers can strictly decide what a role can or cannot. Being a model, GAIA does not propose an implementation of its role system, and thus of its permission logic; this implies that its implementation could be not based on a standard approach, like JAAS. Furthermore, while our approach can be applied to several role systems without requiring changes to those systems, the GAIA approach defines rules

applicable to the GAIA system itself. Finally, it must be noted that, in the GAIA approach, permissions are statically defined, while in our approach, thanks also to the use of the supervisor agent, permissions can be changed depending on run-time constraints.

An approach not tied to the agent world is Locale-BAC (Locale-Based Access Control) [TolGA03], which focuses on the importance of the context where roles are exploited. In fact, it is important to exploit contextual information for authorization decisions, since the context can introduce another level of evaluation. For this reason, this approach does not states what an entity can do or not depending only on the role it is playing, but focuses on the contexts it is active in and how these contexts change during the application live. To reach its aim, this approach extends the RBAC definition [SanCFY96] with a *locale*, a multi-session collaborative environment where entities playing roles must establish a session in order to be active in. Thanks to the locale, the system has an extended view over the whole situation, thus it can make security decisions related to the system life and its evolution. Compared to our approach, Locale-BAC is surely stricter and formal, defining also its own language for role and access specifications, but it does not propose a real implementation that developers can exploit in their applications. Furthermore, it must be noticed that context information are evaluated also in our approach, thanks to the use of the supervisor agent.

7 Conclusions

This paper has presented an approach that exploits the Java Authentication and Authorization System (JAAS) in order to perform security checks on single operations for each role assumed by Java agents. The key idea of this approach is that it is possible to analyze data carried by the agent (contained in a kind of hash-map) in order to extract values suitable for an authentication process. Each role operation that needs authorization must be developed in order to extract that data from the agent credentials, and requiring JAAS to authenticate the agent on the base of that data. The authentication process depends on the callbacks obtained by the role from an authenticator entity, which could be a supervisor agent. Once the authentication has been passed with success, the agent can continue performing the specific role operation.

The advantages of this approach are that it promotes a fine grain control over roles, allowing developers to manage security at a single role operation for each different agent instance (even of the same class). In fact, other approaches adopt either a coarse grain control over roles because of the “group” management performed by the standard Java SecurityManager, or a management at the whole role level, allowing/denying the role assumption. We have integrated this approach into our RoleX role system, obtaining its second version. In our approach, the authentication process is transparent to the agents, which simply use roles; thanks to that, our approach does not require changes to the role system implementation and can be easily integrated into already existing roles without requiring changes to the agents that must use those roles. Moreover, since our approach is based on a component of the Java 2 architecture, it can be integrated in other software systems and can adhere to the security policies promoted by the Java platform.

Future work will deeply explore the problem of the credential acquisition, which actually represents the greatest difficulty of the implementation of this approach. In fact, it could happen that the agent is not carrying all the data required for the authentication; in this case the authentication should proceed basing on other credentials the agent owns. The use of not ad-hoc credentials makes the authentication difficult, leading to a *trust* problem.

References

- [BecGKM99] M. Becht, T. Gurzki, J. Klarmann, M. Muscholl, "ROPE: Role Oriented Programming Environment for Multiagent Systems", Fourth IFCIS Conference on Cooperative Information Systems, Edinburgh (UK), September 1999.
- [CabFL03] G. Cabri, L. Ferrari, L. Leonardi, "Role Agent Pattern: a developer Guideline", in Proceedings of the 2003 IEEE International Conference on Systems, Man & Cybernetics, 5th-8th October 2003, Washington D.C., U.S.A.
- [CabFL03b] G. Cabri, L. Ferrari, L. Leonardi "A Case Study in Role-based Agent Interactions" The IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE-2003), Linz, Austria, June 2003
- [CabFL04] G. Cabri, L. Ferrari, L. Leonardi, "Mailconfigurator: Automatic Configuration of E-Mail Accounts Through Java Mobile Agents", the third International Conference on Principles and Practice of Programming Java (PPPJ), Las Vegas, Nevada, USA (2004)
- [CabFL04c] G. Cabri, L. Ferrari, L. Leonardi, "A Role-based Mobile-Agent Approach to Support E-Democracy", to be published on "Applied Soft Computing", Elsevier
- [CabFL04d] G. Cabri, L. Ferrari, L. Leonardi, "Agent Role-based Collaboration and Coordination: a Survey About Existing Approaches", in the proceedings of the International Conference on Systems, Man and Cybernetics (SMC), The Hague, The Netherlands, (2004)
- [CabFL04e] G. Cabri, L. Ferrari, L. Leonardi, "Embedding JAAS in Agent Roles to Apply Local Security Policies", the third International Conference on Principles and Practice of Programming Java (PPPJ), Las Vegas, Nevada, USA (2004)
- [CabFL05] G. Cabri, L. Ferrari, L. Leonardi, "Injecting Roles in Java Agents Through Run-Time Bytecode Manipulation", IBM Systems Journal, February 2005, Vol. 44 N.1, pp.185-208
- [EJB] SUN Microsystem, "Enterprise Java Beans", available material at <http://java.sun.com/ejb>
- [Fas03] Maria Fasli, "Social Interactions in Multi-Agent Systems: A Formal Approach", The First European Workshop on Multi-Agent Systems (EUMAS), 18-19 December 2003, Oxford, UK
- [GamHJV95] E. Gamma, R. Helm, R. Jhonson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995
- [Jen01] N. R. Jennings, "An agent-based approach for building complex software systems", Communications of the ACM, Vol. 44, No. 4, pp. 35-41, 2001.
- [LucMP03] Michael Luck, Peter McBurney, Chris Preist, "Agent Technology: Enabling Next Generation Computing – A Roadmap for Agent Based Computing", AgentLink, <http://www.agentlink.org/roadmap>
- [San96] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman, "Role-based Access Control Models", IEEE Computer, Vol. 20, No. 2, pp. 38-47, 1996.
- [SUNJA] SUN Microsystems, "Java Authentication and Authorization Service (JAAS)", available material at <http://java.sun.com/products/jaas/>
- [WooJK00] M. Wooldridge, N. R. Jennings, and D. Kinny, "The Gaia Methodology for Agent-Oriented Analysis and Design", Journal of Autonomous Agents and Multi-Agent Systems, Vol. 3, No. 3, pp. 285-312, 2000.
- [XuF03] Cheng-Zhong Xu, Song Fu, "Privilege Delegation and Agent-Oriented Access Control in Naplet", 23rd International Conference on Distributed Computing Systems Workshops (ICDCSW'03), Providence, Rhode Island, USA

- [TolGA03] William J. Tolone, Robin A. Gandhi, Gail-Joon Ahn, "Locale-Based Access Control: placing collaborative authorization decisions in context", in Proceedings of the 2003 IEEE International Conference on Systems, Man & Cybernetics, 5th-8th October 2003, Washington D.C., U.S.A.
- [SanCFY96] Sandhu R. S., Coyne E. J., Feinstein H. L., Youman C.E., "Role-Based Access Control Model", IEEE Computer, 29(2), pp. 38-47, Feb. 1996