JADE SECURITY GUIDE

USAGE RESTRICTED ACCORDING TO LICENSE AGREEMENT.

last update: 28-February-2005. JADE 3.3

Authors: JADE Board

TABLE OF CONTENTS

1 INTRODUCTION

This document describes how to use the JADE-S (Version 2) add-on of the JADE platform that provides support for security in multi agent systems. It should be noticed that this add-on completely replaces and extends the first security add-on (JADE-S version 1) distributed with version 2.61 of JADE. With respect to its previous version, JADE-S Version 2 provides additional features such as end-to-end message signature and encryption, fixes a number of bugs and is compatible with the LEAP add-on, which allows running JADE agents on mobile phones. In the rest of the document the term JADE-S will be used to refer to the Version 2 of the security add-on.

This guide focuses on starting and administrating a secure platform. Please refer to the javadoc for a complete description of the APIs that allow accessing security features from the code.

The version 2 of the JADE-S add-on was developed by the JADE Board and requires the release 3.2 or later of JADE.

1.1 Target Audience

This document is intended for JADE users who are interested in developing multi-agent applications that require some degree of security such as guaranteed message integrity and confidentiality and authorization checks when an agent performs specific actions.

The reader is assumed to be already familiar with JADE. For people new to JADE we strongly recommend to first read the JADE Administrators Guide and Programmers Guide or the JADE Programming Tutorial first available on the JADE web site (http://jade.tilab.com).

1.2 Rationale

Distributed systems when deployed in an open environment require a high level of security at both the infrastructure and application levels. Distributed Multi-Agent systems, leveraging agent's autonomy and mobility, require even greater attention to security issues. Depending on the particular scenario a multi agent system is deployed for, there are a number of security threats that must be taken into account as highlighted by the examples below.

1) Let us consider an agent-based auction system where each participant is represented by an agent that places bids to buy goods on behalf of its user. In this scenario agents compete between each other and a malicious agent could try to kill its competitors to keep the price as low as possible. Without proper security checks, a simple request to the AMS would be sufficient to achieve that.

2) Consider a business application where confidential information must be exchanged between distinct agents. A malicious agent or hacker may sniff the message content or even tamper it to make, for instance, the receiver agent perform an action different than the one it had been originally request for.

3) Finally, consider a data mining application based on mobile agents that move across machines to intensively query a number of databases locally. A malicious agent could go to a remote host and act as a virus deleting or tampering, for instance, relevant information in the operating system of the host machine.

The JADE-S add-on V2 allows protecting a JADE-based multi agent system against security attacks such as those presented in the above examples. More in details all the components (agents, and containers) in a platform are owned by authenticated users, who in turn are authorized by the platform administrator to perform only certain privileged actions. Each agent owns a public and private key pair by means of which it can sign and encrypt messages.

1.3 Current limitations

The Version 2 of the security add-on described in this document still contains some limitations.

- Mobility related permissions are still missing. As a consequence, in order to be secure (according to the security mechanisms implemented so far), a JADE-S V2 based platform should not support agent mobility at all. In order to do that it is sufficient not to start the Agent Mobility Service when launching JADE.
- Pieces of information exchanged between containers (called horizontal commands) are transferred over secure channels (SSL) but are not signed. A malicious agent or hacker could issue a fake horizontal command in order to force a remote container to behave in a way that differs from the original intent.
- Most of the security features are not accessible from agents running on MIDP devices.

Version 3 of the JADE-S add-on providing features addressing the aspects mentioned above is planned for the end of 2004.

1.4 Requirements

In order to use a JADE platform with installed security features, it is sufficient to download JADE and the security add-on. In order to compile them the version 1.6 of ANT is required (http://ant.apache.org).

1.5 Document Layout

Section 2 gives an overview of the three main features provided by JADE-S V2, i.e. user authentication, agent action authorization and message signature and encryption. Section 3. provides step-by-step guidance on how to start and administer a secure JADE platform and how to make agents exchange signed and encrypted messages.

2 JADE SECURITY OVERVIEW

This section provides an overview of the security features supported by JADE-S, namely user authentication, agent actions authorization against agent permissions and message signature and encryption. The details about option names and configuration files are described in the following.

2.1 Authentication

Authentication provides a guarantee that a user starting a JADE platform, and thereby generates containers and agents within that platform, is considered legitimate within the secured scope of the computational system hosting the main container of that platform. *Legitimate* in the case of the JADE authentication process implies that the user is known to the system by having at least one valid identity and associated password.

In general, an authentication system is composed of two main elements: a "CallbackHandler" that allows the user to provide its username and password and a "LoginModule" that checks if these username and password are valid.

The JADE authentication mechanism is based on the JAAS (Java Authentication and Authorization Service, http://java.sun.com/products/jaas) API that enables the enforcement of differentiated access control on system users. The JAAS mechanism provides a set of de facto LoginModules; the **Unix**, **NT** and **Kerberos** modules have been implemented in this release. The Unix and NT modules are Operating System dependent and are designed to use the identity of the user extracted from the current Operating System session. The Kerberos module is system independent in operation, but requires system-specific configuration prior to use. All necessary configuration information that is not detailed in this document can be found on or referenced from the JAAS homepage as indicated above. In addition to these standard modules, a special LoginModule is available termed **Simple**. This allows basic authentication against a plain text password file held in a system folder and is intended primarily for testing purposes.

The current release also provides several callback mechanisms:

- **cmdline** – Username and password are provided as JADE configuration parameter, either as command-line parameter (–owner user:pass), or into the configuration file (owner=user:pass).
- **text** – The starting container prompts the user for username and password insertion.
- **dialog** – The starting container shows a dialog where the user can insert username and password.

If any problem occurs during authentication, or the user fails to be correctly authenticated then the system will exit and generate appropriate error messages.

2.2 Permissions

Thanks to the authentication mechanism described in section 2.1, a JADE-S based platform becomes a multi-user system where all components (containers and agents) are owned by an authenticated user. All actions that agents can perform in the platform can be permitted or denied according to a set of rules. By this way it is possible to selectively grant access to platform services or to application resources. This set of rules is usually described into a file named: policy.txt, which follows the default Java/JAAS syntax, but uses an extended policy model that allows greater flexibility into a distributed agent-based scenario.

Reflecting the JADE architecture that includes a Main Container and several peripheral containers, two types of policy files can be used to grant permissions to agents:

1. The MainContainer policy file that specifies platform-wide permissions such as "Agents owned by user Bob can kill agents owned by user Alice".

2. The peripheral container policy files (one per container) that specify container-specific permissions (such as "Agents owned by user Bob can kill agents owned by user Alice on the local container").

Container policy files also regulate the access to local resources (JVM, file system, network, etc...). A list of permissions that can be used into policies can be found at Annex II, Several examples of policies can be found in this document and into the `'add-ons/security/examples'` directory.

2.3 Message integrity and confidentiality

Signature and encryption guarantee a certain level of security when sending an ACL message both to an agent running on the same or a foreign platform. Signatures are a well-known safeguard to ensure the integrity of a message (confidence that data has not been tampered with during transmission) and the identity of the message originator. Encryption, on the other hand, ensures confidentiality of the message by protecting message data from eavesdropping (confidence that only the intended receiver will be able to read the clear message). As background information, an ACL message is composed of two parts: the **envelope**, which contains transport related information and the **payload**, which contains the actual sensitive data.

Figure 1. Structure of a FIPA ACL Message

In JADE-S "Signature" and "Encryption" always apply to the entire payload in order to protect all the important pieces of information contained in the slots of the ACL message (content, protocol, ontology, etc.). The security-related information (such as the signature, the algorithm or the key) is then placed into the envelope. As shown in the `messaging.SecureSenderAgent` and `messaging.SecureReceiverAgent` included in the examples directory, users do not need to deal with the actual signature and encryption mechanisms, but just need to request a message to be signed or check whether a received message has been signed. Note that if some problems occur whilst signing, encrypting, verifying or decrypting a message, the message is discarded and a failure notice is returned to the sender as a FAILURE message from the AMS. The immediate

implication is that each time an agent receives a signed message, the signature is valid (i.e. the agent does not need to check the signature again).

3 HOW TO USE JADE-S V.2

In conformance with the new JADE kernel architecture based on the concept of *service*, each one dealing with a specific aspect, the security support is implemented as a set of JADE Services. Regarding security there are four relevant services:

- `jade.core.security.SecurityService`: This service is in charge of the authentication mechanism and also provides all functionality common to all security related services such as crypto engines and management of agent key pairs.

- `jade.core.security.permission.PermissionService`: This service is in charge of checking that agents performing actions such as sending messages, moving to other containers and requesting the AMS to create/kill other agents are actually authorized to do that.

- `jade.core.security.signature.SignatureService`: This service is in charge of signing messages when requested by the sender and verify the validity of incoming signed messages.

- `jade.core.security.encryption.EncryptionService`: This service is in charge of encrypting messages when requested by the sender and decrypt incoming encrypted messages.

Therefore, in order to start a JADE-S based secure platform it is necessary to launch JADE activating the **SecurityService** and, depending on the application needs, the **SignatureService**, the **EncryptionService** and/or the **PermissionService**. As an example, the command line below starts a platform where agents can send signed messages (see also the JADE Administrator's Guide for a detailed explanation on activating JADE services).

```
java -cp <jade-classes>;<jade-s-classes> jade.Boot -gui -services
jade.core.security.SecurityService;jadecore..security.signature.SignatureService
```

3.1 Starting and administrating a multi-user platform

This section provides step-by-step guidance on how to start and administrate a secure platform where each component (container and agents) is owned by an authenticated user. This is done by illustrating the Startup example included in the examples.Startup directory. In this example two users "alice" and "bob" own the Main Container and a peripheral container respectively and some agents on them. They and their agents are granted with different permissions (see 3.1.1) and as a consequence certain actions will be forbidden. We use the Simple LoginModule for authentication so that no special system configuration is required to run the example. As mentioned in 2.1 this LoginModule stores valid usernames and passwords in a text

file in clear and is there only for testing and exemplifying purposes. The content of the password file `examples/startup/main/passwords.txt` is reported below. User information is provided in the form <username> <password> with each user information on a separate line.

```
alice alice

bob bob
```

3.1.1 **Assigning permissions**

For assigning rights the JAAS policy file syntax is used. *Rights* can be granted to code, users and agents and are identified by a principal of type `jade.security.Name`. The policy file for this example has to grant several privileges in order to run the examples. Firstly the JADE code has to be granted all JAVA security permissions:

```
grant codebase
 "file:<JADEROOT>/add-ons/security/lib/JadeSecurity.jar" {

   permission java.security.AllPermission;  };

grant codebase "file: <JADEROOT>/lib/jade.jar" {

   permission java.security.AllPermission;};

grant codebase "file: <JADEROOT>/lib/jadeTools.jar" {

   permission java.security.AllPermission;  };
```

Additionally the user “alice” has to be granted several privileges. Below the policy required to create/kill platforms, containers and agents.

```
grant principal jade.security.Name "alice" {
  permission jade.security.PlatformPermission "", "create,kill";
  permission jade.security.ContainerPermission "", "create,kill";
  permission jade.security.AgentPermission "", "create,kill";
  permission jade.security.AgentPermission "", "suspend,resume";
  permission jade.security.AMSPermission "", "register, deregister,
modify";
  permission jade.security.MessagePermission "", "send-to";
};
```

The user “bob” is granted the permissions to create a simple container, create agents owned by bob on containers owned by bob and whose name starts with “bob-“ All agents running under the user “bob” are granted the right to send messages to all other agents in the container. Additionally “bob” is granted the permission to register and deregister agents with the AMS.

```
grant principal jade.security.Name "bob" {
  permission  jade.security.ContainerPermission  "container-owner=bob",
"create, kill";
  permission jade.security.AgentPermission "agent-owner=bob,container-
owner=bob,agent-name=bob-*", "create";
  permission      jade.security.AgentPermission      "agent-owner=bob",
"kill,suspend,resume";
  permission   jade.security.AMSPermission          "agent-owner=bob",
"register,deregister,modify";
  permission jade.security.MessagePermission "", "send-to";
};
```

3.1.2 **Starting the main container**

Having configured the password and policy file we can now start the Main container owned by user "alice". All JADE related parameters can either be passed as command line options or via configuration file. In this example, since there are several parameters, we use the `examples/startup/main/main.conf` configuration file.

An explanation of the security-related configuration parameters follows.

Security services

```
services = jade.core.security.SecurityService;\
jade.core.security.permission.PermissionService;\
jade.core.security.signature.SignatureService;
```

In order to exploit the security support the SecurityService, PermissionService and SignatureService must be activated. It should be noticed that requests (e.g. to create or kill other agents) to the AMS are always automatically signed so that the AMS can be sure about the identity of the requester. The EncryptionService is not required unless agents need to exchange confidential messages (see 3.2) and therefore is not activated in this example.

Policy file

```
java.security.policy = policy.txt
```

This parameter indicates the location of the policy containing permission information. Note that this is a JAVA and not a JADE specific parameter. In this case, the policy.txt file described in 3.1.1 is indicated.

Login module

```
jade.security.authentication.loginmodule = Simple
jade.security.authentication.loginsimplecredfile = passwords.txt
```

As already mentioned we use the Simple LoginModule and the passwords.txt password file.

In addition to the above parameters the JAAS framework requires a proper configuration regarding which login module to use.

```
java.security.auth.login.config=jaas.conf
```

Within the JAAS configuration file the following line has to be added defining the login module to be used for simple authentication.

```
Simple {
   jade.core.security.authentication.SimpleLoginModule required;
};
```

Callback handler

```
jade.security.authentication.logincallback = Cmdline
```

This parameter indicates the callback handler in charge of retrieving the username and password of the user that is starting this Main Container. Since we specify CmdLine the username and password are given as configuration parameters as well. Note the syntax <username>:<password>

```
owner = alice:alice
```

Since user "alice" owns this Main Container, agents activated at bootstrap time are owned by user "alice" as well.

Finally moving to the examples\startup\main directory and typing

```
java <jade-libs>;<jade-s-lib> jade.Boot -conf main.conf
```

starts the Main Container owned by user "alice".

3.1.3 **Starting an agent**

By specifying in the configuration file to start a JADE RMA called alice-rma as below,

```
agents=alice-rma:jade.tools.rma.rma
```

The JADE GUI should appear as in Figure 2. Note that "alice" appears in the owner column in the right part of the GUI when selecting an agent.

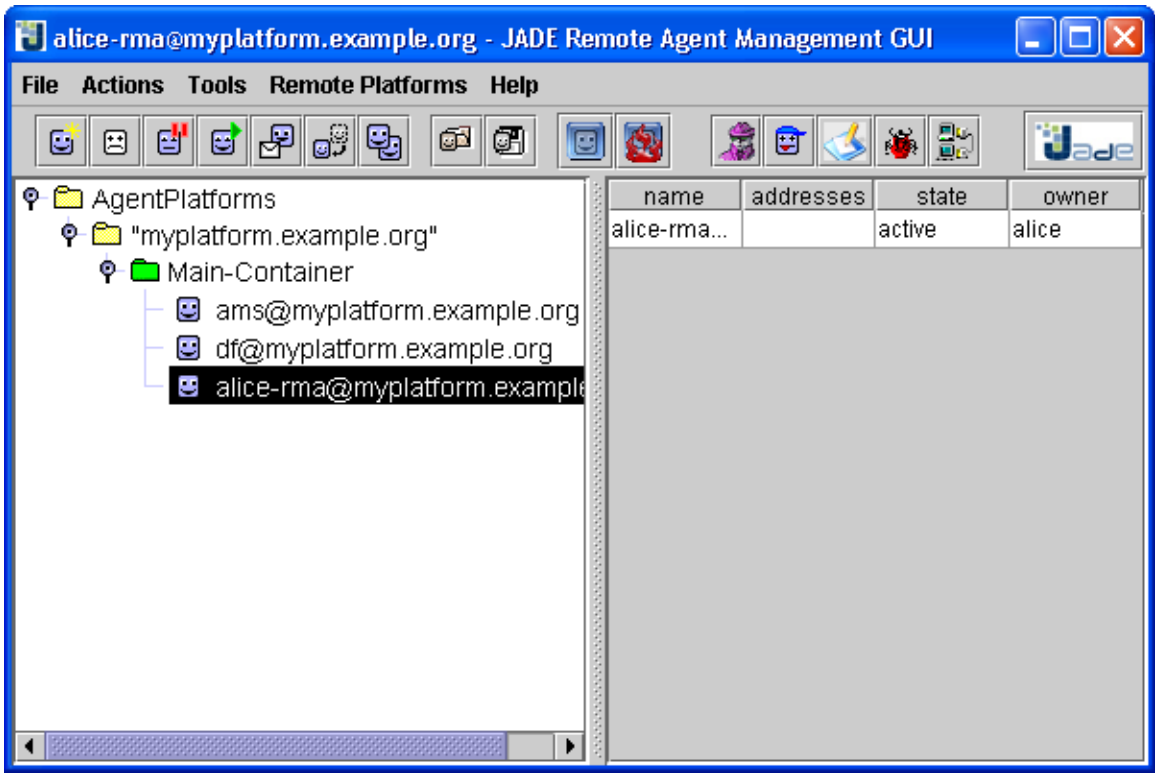


Figure 2. The JADE GUI

3.1.4 **Attaching a container**

In this section the necessary steps to attach a container to a JADE agent platform will be shown. The files for the example can be found in the directory `examples/startup/cont-1`.

For the container, which has to be started, a set of configurations has to be done. The configuration file contains the services that have to be started. `services=\`

```
jade.core.security.SecurityService;\

jade.core.security.permission.PermissionService;\

jade.core.security.signature.SignatureService;\

jade.core.event.NotificationService



java.security.policy=policy.txt
```

Now a policy file, which is specific for this container, has to be created. Here again the

permission grants for the JADE source code have to be made explicit. Moreover the full set of permission are granted to user bob. Additionally the generic principal "*" (i.e. everyone) is granted with the right to send message.

It should be noticed that the container policy file is only valid locally.

```
grant codebase

 "file:<JADEROOT>jade/add-ons/security/lib/jadeSecurity.jar" {

   permission java.security.AllPermission;  };

grant codebase "file: <JADEROOT>/lib/jade.jar" {

   permission java.security.AllPermission;};

grant codebase "file: <JADEROOT>/lib/jadeTools.jar" {

   permission java.security.AllPermission;  };



grant principal jade.security.Name "bob" {

   permission jade.security.AgentPermission "",
   "create, kill";

   permission jade.security.AgentPermission "",
   "suspend, resume";

   permission jade.security.AMSPermission   "",
   "register,deregister,modify";

};



grant principal jade.security.Name "*" {

   permission jade.security.MessagePermission "",
   "send-to";

};
```

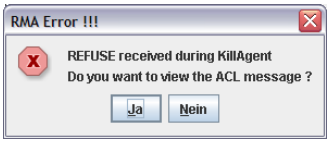
A new JADE container can be started by specifying the following command line arguments:

```
java -jade.Boot -conf cont-1.conf
```

After executing this command, since the Dialog callback handler is specified in the cont-1.conf file, a window should appear requesting the user to enter username and password. Now when entering the user information of "bob", the container will start up.

3.1.5 **Authorization**

In the illustrated example, user "alice" has full platform wide permissions, but is not granted any permission on container cont-1 (a part from the right of sending messages). As a consequence, using alice-rma it will be possible to create and kill agents owned by alice or bob on the main container. When trying to create an agent on container cont-1 on the other hand, and authorization error should occur and the following dialog window should appear..



Similarly, since bob is only granted the right to create agents owned by bob on containers owned by bob and whose name starts with "bob-", trying to create an agent that does not have all these characteristics (e.g. an agent called "a" or an agent on the main container that is owned by alice) should result in the same error.

3.2 Exchanging signed and encrypted messages

Since signing and/or encrypting a message clearly slow down the performances of agent communication, messages are neither encrypted nor signed by defaults. It is the responsibility of the agent sending a message to explicitly request the platform to sign and/or encrypt a message. This is done by means of the `setUseSignature`() and `setUseEncryption`() methods of the `SecurityHelper` that can be retrieved (as for all ServiceHelpers) by means of the `getHelper`() method of the `Agent` class.

The example included in the examples.messaging directory shows in details how to send signed and/or encrypted messages. More in details the `SecureSenderAgent` and `SecureReceiverAgent` show the sending part and the receiving part respectively and should be used together. In order to start them just launch a JADE main container with the security service, signature service and encryption service and specifying the SingleUser authentication type as below.

```
java -cp <jade-classes>;<jade-s-classes>;<example-classes> jade.Boot -gui
-services
jade.core.security.SecurityService;jade.core.security.signature.SignatureService;
jade.core.security.encryption.EncryptionService
-jade_security_authentication_loginmodule SingleUser s:messaging.SecureSenderAgent
r:messaging.SecureReceiverAgent
```

It should be noticed that the same code works also in the case that the receiver lives in a remote platform.

ANNEX I: CONFIGURATION PARAMETERS

Parameter Name:	services
Synopsis	Services to be started by the JADE platform.
Values	jade.core.security.SecurityService jade.core.security.permission.PermissionService jade.core.jade.core.security.signature.SignatureService jade.core.security.encryption.EncryptionService

Please note that when specifying services as start up parameter, the services, which are installed by default, have to be installed manually if needed. These are:

- jade.core.mobility.AgentMobilityService;
- jade.core.event.NotificationService

Parameter Name:	jade.security.policy
Synopsis	Location of the policy file containing permissions.
Values	*policy.txt (default value)*

Parameter Name:	jade.security.authentication.logincallback
Synopsis	Method that is used to specify how a user can authenticate.
Values	Cmdline user information is specified as a command line parameter of jade.Boot Text prompt for the user to enter information *Dialog* show a dialog window to ender user information (default value)

Parameter Name:	Owner
Synopsis	When jade.security.authentication.logincallback is specified with the CmdLine option this parameter is used to specify the user
Values	User information in the form <username>:<password>.

Parameter Name:	jade.security.authentication.loginsimplecredfile
Synopsis	Path to the file where to find credentials when using simple Authentication
Values	*passwords.txt (default value)*

Parameter Name:	jade.security.authentication.loginmodule
Synopsis	Select which JAAS or custom login module will be used for verifying user's password.
Values	*Simple (default value)* NT Unix Kerberos

Parameter Name:	jade.security.AsymAlgorithm
Synopsis	The default asymmetric algorithm that is used to generate a key pair.
Values	*RSA (default value*) DSA …

Parameter Name:	jade.security.AsymKeySize
Synopsis	Sets the default key size for public/private keys.
Values	*512 (default value)* 1024 2048

Parameter Name:	jade.security.SymAlgorithm
Synopsis	Sets the default symmetric algorithm used when the message has to be encrypted
Values	*AES (default value)* Blowfish DES DESede TripleDES PBEWith<digest>And<encryption> e.g. PBEWithMD5AndDES PBEWith<prf>And<encryption> e.g. PBEWithHmacSHA1 AndDESede

Parameter Name:	jade.security.SymKeySize
Synopsis	Sets the default key size of the secret key used when a message has to be encrypted
Values	**DES**: keysize must be equal to 56 **TripleDES**: keysize must be equal to 112 or 168 **AES**: *128 (default value)*, 192, 256 **Blowfish**: keysize must be a multiple of 8, and can only range from 32 to 448, inclusive

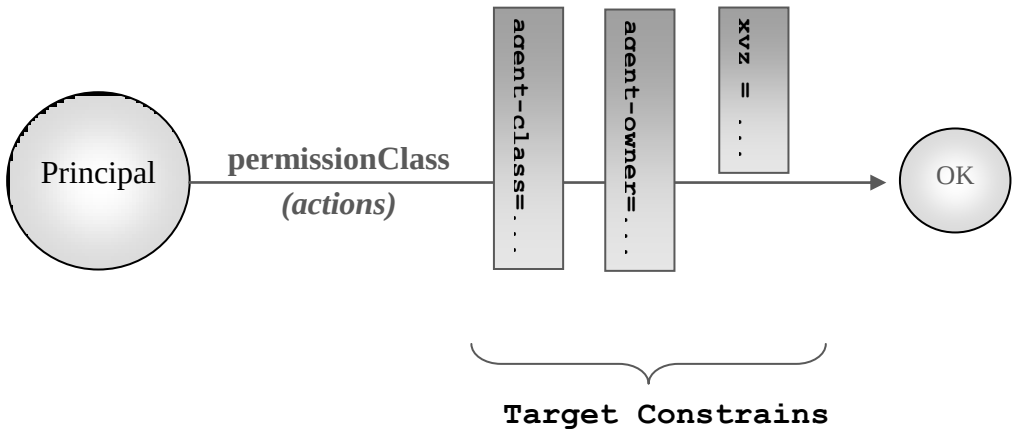
Parameter Name:	jade.security.SignAlgorithm
Synopsis	Sets the default algorithm that is used when messages have to be signed.
Values	SHA1withRSA MD5withRSA SHA1withDSA DSA …

ANNEX II: PLATFORM SERVICES PERMISSIONS

Permission	actions	Target Constrains
jade.security.AgentPermission	create kill	agent-owner agent-name container-owner agent-class *(local container only)*
	suspend resume	agent-owner agent-name agent-class *(local container only)*
jade.security.MessagePermission	send-to	agent-owner agent-name
jade.security.PlatformPermission	create kill	
jade.security.ContainerPermission	create kill	container-owner

Policy Syntax:

```
grant principal jade.security.Name "<principalName>" {
   permission <permissionClass> "<targetConstrains>",
                                "<actions>";
};
```



Example #1:

```
grant principal jade.security.Name "alice" {
   permission jade.security.ContainerPermission "", "create";
   permission jade.security.AgentPermission "agent-class=jade.core.Agent",
                                            "create,kill";
   permission jade.security.AMSPermission   "agent-class=*, agent-owner=*",
                                            "register,deregister,modify";
   permission jade.security.MessagePermission "agent-owner=alice",
                                              "send-to";
};
```

Alice can create remote containers, can create and kill any agent of type: '`jade.core.Agent`'. She (and her agents) also has the permission to send messages to agents owned by "alice".