Technical University of Vienna
Information Systems Institute
Distributed Systems Group

A flexible and extensible security framework for Java code

Manfred Hauswirth, Clemens Kerer
and Roman Kurmanowytsch
M.Hauswirth@infosys.tuwien.ac.at
C.Kerer@infosys.tuwien.ac.at
romank@infosys.tuwien.ac.at

TUV-1841-99-14 Nov 22, 1999

Any piece of code which is run on a computer system can potentially threaten the security, privacy, and integrity of the system and its users. This truism has gained new importance with the introduction of mobile code systems such as Java applets and mobile agent platforms, in which code may be loaded from outside sources on-the-fly and executed in the user's environment. Thus mobile code systems must provide a security architecture which tries to protect users from erroneous or malicious code. Java as the most popular platform for mobile code includes such an architecture which unfortunately is complicated and tedious to use, lacks sufficient support for system-wide security policy maintenance, has no concepts for user and group security profiles, and does not protect users from misconfigurations or introduction of security holes. This paper describes Java Secure Execution Framework (JSEF) which provides a highly configurable security environment that addresses these drawbacks while remaining compatible with Java's model. JSEF has a hierarchical group concept that supports the definition and propagation of access policies, offers additive and subtractive permissions and policy exceptions, and supports system-wide security policies that users must adhere to but can tailor to their needs. At runtime JSEF enforces the security policy and supports (interactive) security negotiation in case of insufficient privileges. The user can configure JSEF via a set of graphical tools and define security policies which are represented as XML documents.

Keywords: Java security, security management, mobile code

Argentinierstr. 8/184-1
A-1040 Vienna, Austria
phone: +43 1 58801-18402
fax: +43 1 58801-18491
URL: http://www.infosys.tuwien.ac.at/

```
Submitted to the
   9th International World Wide Web Conference (WWW9)
   The Web: The Next Generation
   Amsterdam, May 15 - 19, 2000
```

A flexible and extensible security framework for Java code

Manfred Hauswirth, **Clemens Kerer**, and **Roman Kurmanowytsch**

Distributed Systems Group,
Technical University of Vienna,
Argentinierstraße 8/184-1, A-1040 Vienna, Austria
http://www.infosys.tuwien.ac.at/

Abstract

Any piece of code which is run on a computer system can potentially threaten the security, privacy, and integrity of the system and its users. This truism has gained new importance with the introduction of mobile code systems such as Java applets and mobile agent platforms, in which code may be loaded from outside sources on-the-fly and executed in the user's environment. Thus mobile code systems must provide a security architecture which tries to protect users from erroneous or malicious code. Java as the most popular platform for mobile code includes such an architecture which unfortunately is complicated and tedious to use, lacks sufficient support for system-wide security policy maintenance, has no concepts for user and group security profiles, and does not protect users from misconfigurations or introduction of security holes. This paper describes Java Secure Execution Framework (JSEF) which provides a highly configurable security environment that addresses these drawbacks while remaining compatible with Java's model. JSEF has a hierarchical group concept that supports the definition and propagation of access policies, offers additive and subtractive permissions and policy exceptions, and supports system-wide security policies that users must adhere to but can tailor to their needs. At runtime JSEF enforces the security policy and supports (interactive) security negotiation in case of insufficient privileges. The user can configure JSEF via a set of graphical tools and define security policies which are represented as XML documents.

1 Introduction

Mobile code denotes code that traverses a network and executes at a remote site. The process of traversing can either be active as in the case of mobile agents which move around in a network at their own volition, or it can be passive, i.e., a party at a remote location initiates the transmission of the code to the remote site and its execution there. Java is the most popular platform which supports such code mobility in a platform-independent way.

In conjunction with the Internet this opens a wealth of new possibilities for the development of general purpose, portable software, software architectural styles, and software deployment. Unfortunately, this also increases security problems to the same extent as it supports new software paradigms. For example, downloaded code can include a virus or be a Trojan horse and thus pervert the concept of mobility over the Internet to threaten computer systems in a way not known before. As any mobile code platform Java suffers from four basic categories of potential security threats [2, 8, 17, 18, 19]:

> **Leakage:** unauthorized attempts to obtain information belonging to or intended for someone else

Tampering: unauthorized changing (including deleting) of information
Resource stealing: unauthorized use of resources or facilities (e.g., memory, disk space)
Antagonism: interactions not resulting in a gain for the intruder but annoying for the attacked party.

To deal with these threats Java provides a special runtime environment that tries to protect users from erroneous or malicious mobile code and tries to ensure the integrity, security, and privacy of the user's system. It provides good protection against the two most dangerous threats of leakage and tampering, while the less dangerous ones of resource stealing and antagonism cannot be fully prevented. This, however, is due to the fact that it is hard to distinguish automatically between legitimate and malicious actions.

Java's security architecture offers many low-level security mechanisms, e.g., access permissions on resources, but falls short when it comes to higher-level security management and maintenance such as hierarchical policies or user groups. Standard Java security is complicated and tedious to maintain, it does not support flexible system-wide security policies and lacks concepts for defining security profiles. It has no notions of users and groups and provides only very limited means for hierarchically organized security configurations. Lack of such higher-level concepts complicates tailoring of security requirements to the needs of a specific user and may easily cause misconfigurations or introduction of security holes.

In this paper we present the Java Secure Execution Framework (JSEF) which solves these shortcomings. JSEF (pronounced Josef) provides a hierarchical security policy scheme which supports both local, user-specific security policies and a global security policy defined by the administrator. It supports the definition of user groups which can be freely structured into a hierarchy and assigned security policies. A user can be member of a set of groups with different security profiles which aids administrators in the definition, assignment, and maintenance of security policies for a user or a group of users. Additionally, JSEF offers useful functionalities beyond Java's capabilities: Policy and group definitions are represented as XML documents; policy, configurations, and mobile code can be retrieved from arbitrary locations; and security issues can be negotiated interactively at runtime. JSEF is based on the Java 2 security architecture and is fully compatible with it. It can be used with any Java code and in any environment which is compatible with Java 2, for example, in Java-based mobile agent systems or for extended Java security features in Web browsers. JSEF originally was developed as part of the Minstrel push system project **[11, 21]** to provide a flexible security environment for executable channel content (so-called pushlets) and agents **[10]**.

This paper is organized as follows: **Section 2** gives a concise overview of Java's security model as a prerequisite for the following discussions. On the basis of this description **Section 3** discusses Java's security model and points out its shortcomings which provided the motivation and requirements for JSEF. **Section 4** then presents JSEF's security model, its concepts, and processes. Some interesting parts of JSEF's implementation are described in **Section 5** as well as the problems we had to cope with during coding which provided many further insights into Java's security. In **Section 6** we overview the main tools developed for JSEF which offer easy-to-use access, configuration, and management functionalities for users and administrators. **Section 7** presents work related to JSEF. We summarize and give our conclusions in **Section 8**. JSEF and all associated tools are available at **http://www.infosys.tuwien.ac.at/Minstrel/JSEF/**.

2 Java's Security Architecture

Several approaches have been devised to cope with the security implications of mobile code. According to **[22]** four practical techniques for securing mobile code exist:

Sandbox model:
This model restricts the privileges of the code to a limited set of operations.
Code signing:
Only code which comes from trusted sources is granted full access.
Firewalling:
When code enters a trusted domain it is examined; the decision whether and how to run it is based on specific code properties such as the origin or special behavioral properties.
Proof-carrying code:
The mobile code carries a proof that it satisfies certain properties; this is a promising new technique that can currently be applied only in very limited settings (e.g., some programs can proof that they do not contain buffer overflows).

Java uses a hybrid approach that combines *sandboxes* and *code signatures*. The sandbox approach in Java is implemented by providing access to system resources exclusively via the Java core classes which are part of the Java distribution and trusted per default. Those classes act as a security shield and grant or forbid access to resources based on a security policy which in turn depends on the origin of the code and an optional signature. Every time the Java Virtual Machine (JVM) **[15]** runs a piece of code - a class file in Java's terminology - and the Java security mechanisms are in place, the following steps occur **[7]**:

1. The Java virtual machine obtains a class file and accepts it if the file passes preliminary bytecode verification **[25]**.
2. The Java virtual machine determines the class's code source. This step includes signature verification, if the code appears to be signed.
3. The Java virtual machine consults the security policy and composes the set of permissions to grant to this class. In this step, the policy object will be constructed, if it has not been already.
4. The Java virtual machine loads and defines the class, and marks the class as having been granted the set of permissions.
5. The Java virtual machine instantiates the class into objects and executes their methods. Runtime type-safety check continues.
6. If at least one method of a class is in the call chain when a security check is invoked, the access control code examines the class's set of granted permissions. It does this to see if there is sufficient permission for the requested access. If yes, the execution continues. If no, a security exception occurs. When a security exception - which is a runtime exception - occurs and is not caught, the Java virtual machine aborts.
7. When the class file and the instantiated objects are no longer in use, they are garbage-collected.

In **step 1** the **Verifier** performs a set of security checks before a class is loaded. These checks guarantee important properties such as the correct class file format, correct parameter types, and binary compatibility with other class files. The purpose of these checks is to enhance performance since otherwise all these checks would have to be performed during runtime. In addition, they assure that no malformed class is loaded into the Java runtime environment. Two crucial requirements for the proper functioning of the Verifier are that the Verifier is correctly embedded into the Java runtime system and that the Verifier is implemented correctly. A correct embedding assures that no class can be loaded without being checked by the Verifier and the correct implementation guarantees that only valid Java classes can pass the Verifier. Both requirements are not formally proven, thus, you have to rely on and trust SUN's implementation. All application classes apart from the Java core classes which are trusted by default are subject to the Verifier **[25]**.

If a class has passed the Verifier, the *class loader* loads the class, checks optional signatures, and defines the class. **Step 2** constructs the class's code source which consists of the location from which the class was obtained and a set of certificates representing an optional signature. Java supports digitally signed classes only if they are encapsulated in a JAR file and verifying a digital signature is part of the process of constructing a class's code source.

The class's code source, i.e., its origin and its signature, are the key input for the security policy construction for a given class in **step 3**. In Java 2 the security policy is defined in terms of *protection domains* which define what a piece of code with a given code source is allowed to do. Hence, a protection domain contains a code source with a set of associated permissions. This concept is powerful since privileges can be granted on a fine-grained level (e.g., access to a single file or port can be controlled) and wildcards can be used both in the code source as well as in the permission definition to simplify the process of specifying security policies. Given the code source of a class, the security policy (i.e., a collection of protection domains) is searched to collect all the permissions applying to the class.

After having determined a class's code source and permissions the class loader finally defines the class in **step 4**. Defining a class makes it publicly available and adds it to the class loader's cache of classes. Keeping a cache of already defined classes is important to avoid the use of different implementations of the same class which would result in problems with class variables and methods or worse.
In fact, two classes are considered equal if and only if they have the same name and were loaded by the same class loader. As a consequence, two applets which use classes with the same name remain separated since each applet is loaded by its own class loader and, thus, the classes are not considered equal. Every applet codebase has a dedicated class loader.
Furthermore, the *delegation model* for class loaders has to be obeyed. Since class loaders are Java classes, too, they also need a class loader to be loaded and instantiated which results in a hierarchical structure of class loaders. A class loader's defining class loader is called its *parent class loader*. If a class loader fails to find a requested class in its cache, it has to delegate the request to its parent class loader before trying to load the class from any other location. This, again, assures the uniqueness of classes. The obvious question to be answered here is how the first class loader is loaded into a Java runtime system. This is the task of the *primordial class loader* which bootstraps the system and loads the first class loader's class. Apart from storing the class in its cache, the class loader also associates a protection domain with the class reflecting all its permissions. Once a class has been loaded and defined by a class loader all future requests are satisfied from the class cache (**step 5**).

Step 6 decides whether access to a resource is granted. As mentioned above, access to system resources is only possible using Java core classes which in turn are responsible for determining whether access should be granted or denied. In fact, all these core classes contact the *security manager* class to decide how to proceed. If the method call to the security manager returns silently, the requesting caller has enough permission to access the resource and the execution continues. If not, a security exception is thrown and has to be handled by the caller or otherwise the Java virtual machine terminates.

The remaining question is how the security manager decides whether access to a resource is granted. In prior releases of the Java platform this was implemented in the security manager. Since Java 2, however, the security manager is mainly included for compatibility since it delegates nearly all of its tasks to the *access controller* class. This class uses a *stack inspection algorithm* and the security policy to decide how to proceed. The stack inspection algorithm is based on the call stack of the current method, i.e., the (class) history of the request for a given system resource. Since every class was defined with an appropriate protection domain in **step 4**, the stack inspection algorithm can determine the permissions of a class found on the call stack. The basic rule for Java's stack inspection algorithm is that access to a resource is granted if all classes on the current call stack have the permission to access the resource. If, however, a single class contained on the stack does not have the required permission, access is denied. Unfortunately, there is an important exception to this rule which complicates the process. In some cases a more privileged class (e.g., a class belonging to the Java core classes) is executed on behalf of a less privileged class. According to the above rule, the privileged class would be restricted to permissions of the less privileged one which would be inadequate. Imagine an untrusted applet that asks a trusted browser class to obtain the version of the browser it is running in. The browser class might need to read the browser's version from a configuration file or database but would not be allowed to do so because the applet class is not allowed to. To solve this problem, a *privileged mode* was added to the stack inspection process. In the browser example the browser would switch to privileged mode execution to read the configuration entry. The introduction of privileged mode does not bear security risks since no class can gain more privileges than it was originally granted. It is, however, possible for any class to fully exploit its permissions (unrestricted by any less privileged calling classes) to fulfill its task. The stack inspection algorithm returns successfully if either all classes on the stack are granted the requested permission or if a class in privileged mode is detected and all classes which have been called by the privileged one are granted the requested permission. Garbage collecting a class and its objects terminates the life-cycle of a class (**step 7**).

On the basis of these concepts and processes, Java can prevent many security attacks but not all. As already mentioned the Java security model can only be applied to Java code. Security threats that are outside the Java environment cannot be controlled or even noticed by Java's security mechanisms. Only local (to the executing site) resources can be protected by these mechanisms, and security misconfigurations by the user that open security holes cannot be prevented. A more exhaustive description of the features and functioning of Java's security architecture can be found in **[14]** . Aside from the problems discussed above Java does quite a good job. It provides good protection against the two most dangerous threats of leakage and tampering, while the less dangerous ones of resource stealing and antagonism cannot be fully prevented. This, however, is due to the fact that it is hard to automatically distinguish between legitimate and malicious actions.

3 A critical View on Java's Security Model

Although Java's security model provides strong mechanisms to protect the user from security threats, it falls short when it comes to higher-level security configuration and its management. Java's current security model only supports explicit specification of accesses that are permitted. This enables the user to specify all that is necessary to secure his/her site. It is not very practical, however, if the user needs a sophisticated security policy since Java only supports the specification of permitted accesses. Instead of specifying what is permitted, frequently the opposite semantic is required, i.e., to specify what is *not* permitted. JSEF supports both ways of specification by its so-called additive and subtractive permissions.

The current security model of Java uses a two-level configuration approach. A global policy file holds the default permissions for any user on a specific site and a user's local policy file can specify additional permissions. Since Java's security model only supports additive policies, only two extremes for meaningful security configuration exist: Either each user must maintain a private security policy file, or a global policy is specified and user-specific configurations are ignored. Either way has its shortcomings. With the first strategy users can easily introduce security holes - regardless of whether a global policy file exists or not since the user's local policy can extend the global policy in any way - but can have a personalized configuration. In the second case the administrator has total control over the security policy but cannot tailor it to specific users' needs.

JSEF overcomes these problems by providing a hierarchical security policy scheme which supports both local, user-specific security policies and a global security policy defined by the administrator which takes precedence over user policies. At runtime of a Java program, a user's actual policy is defined by merging the user's local policy with the global policy. The user's policy, however, cannot circumvent restrictions imposed by the administrator in the global policy. This scheme is an attempt to improve the management of security policies in a secure yet flexible way which will be explained in detail in **Section 4**. **Moreover**, the Java security model lacks support for user groups while JSEF supports the definition of hierarchical user groups with assigned security policies. A user can be member of several groups that have different security profiles. With user groups being supported, an administrator can easily define a set of profiles in terms of groups and assign these profiles to users depending on the users' requirements. Additionally, these groups can be freely structured into a hierarchy, which further supports maintenance and tailoring of the security policy.

JSEF supports the retrieval of policy definitions from arbitrary sources. It currently uses files (which hold the necessary definitions represented in XML **[1]**), but can easily be tailored to load the policy definitions from other sources such as databases or remote locations. This functionality is facilitated by specialized handler classes provided by the user of JSEF. Also mobile code that is to be executed, can be loaded from arbitrary sources. A sample prototype for loading class files and JAR files from any location in the file system (not only from within the defined Java classpath as in the case of standard Java) is included in JSEF. This concept supports storing of mobile code in arbitrary locations and formats, for example, in a database.

In the standard Java security model the requester of an operation receives a security exception whenever an access is denied by the user's security policy. This typically terminates the execution. If this was not intended, the user has to exit the program that wanted to perform the access, set the appropriate permissions, restart the program, and retry its execution. This can be tedious and time-consuming especially for applets and mobile agents. JSEF therefore provides a security negotiation facility: If a forbidden operation is attempted, JSEF intercepts it *before* the actual access and starts a negotiation process. Currently this means that the user is asked via a GUI whether the access should be denied, permitted once, for the current session, or always and thus be entered into the user's security policy. This supports runtime management of the security policy while still ensuring that the existing policy settings are not violated. The interactive negotiation scheme can also be used as a blueprint for other (semi-) automatic negotiation schemes.

4 The Java Secure Execution Framework Model

To remain compatible with Java's default security model **[3**, **5**, **7**, **24]**, JSEF's policy model is based on a subclass of the Java policy class. Applications that rely on JSEF ignore the settings defined in the Java policy of a user (in the system policy and user policy files) and rely solely on the policy defined for JSEF. JSEF's policy model includes *enhanced policy semantics*, a separation of *local and global policy* settings, and a *dynamic policy negotiation* component. As already mentioned above, policy handler classes are used to abstract from physical storage and support the loading of policy definitions from arbitrary locations with arbitrary protocols.

4.1 JSEF's Enhanced Policy Semantics

4.1.1 Subtractive Policy

JSEF introduces the notions of *additive* and *subtractive permissions*. Additive permissions are the class of permissions as used by the Java security model: They grant permission to access a resource. Subtractive permissions are defined in a similar way but specify which resources must not be accessed. As with additive permissions, subtractive permissions are grouped in (subtractive) protection domains to associate a code source with a set of permissions. The collection of additive protection domains defines the user's *additive security policy* and the collection of subtractive protection domains defines his/her *subtractive security policy*. The subtractive policy always overrules the additive policy. This means that whenever an action is granted and at the same time is forbidden, it is denied. Hence, to access a resource, the following conditions must hold: (1) access to the resource is not forbidden by the subtractive policy and (2) the access to the resource is explicitly allowed by the additive policy. If an action is neither forbidden nor explicitly allowed, JSEF sticks to the semantics of Java's default model and prohibits the action.

Figure 4.1 shows a sample JSEF policy definition of the user *Charly Brown*. In this example an additive protection domain that grants all permissions to all code from *www.sun.com* signed by *CK* is defined. The subtractive part of this policy definition, however, forbids write and execute access to the user's home directory hierarchy for such code.

```
<?xml version="1.0"?>
<!DOCTYPE localPolicy SYSTEM "localPolicy.dtd">
<localPolicy userName = "Charly Brown" lastChanged="10/21/1999">
  <addItems>
    <policyItem signedBy="CK" codeBase="http://www.sun.com/-">
      <permission class="java.io.AllPermission">
      </permission>
    </policyItem>
  </addItems>
  <subItems>
    <policyItem signedBy="CK" codeBase="http://www.sun.com/-">
      <permission class="java.io.FilePermission">
        <permissionName name="/home/-"/>
        <actions name="write execute"/>
      </permission>
    </policyItem>
  </subItems>
</localPolicy>
```

Figure 4.1: A sample policy definition in JSEF including additive and subtractive permissions

The XML DTDs for this and the following JSEF configuration examples are given in the **appendix**. JSEF generally uses XML-based definitions.

4.1.2 Policy Exceptions

Policy Exceptions are another extension to Java's security architecture. which is applied in conjunction with the wildcards `'*'` and `'-'`. `'*'` refers to all files in a given directory, to all ports, or to all hosts depending on which kind of permission is being defined. `'-'` is mainly used with files and code bases to grant access to all files in the given directory and recursively in all its subdirectories. Policy exceptions facilitate an *except-for* semantics. For example, a user can grant access to all files in a given directory except for those listed in the policy exception. **Figure 4.2** defines the same semantics as **Figure 4.1** but uses policy exceptions.

```
<?xml version="1.0"?>
<!DOCTYPE localPolicy SYSTEM "localPolicy.dtd">
<localPolicy userName = "Charly Brown" lastChanged="10/21/1999">
  <addItems>
    <policyItem signedBy="CK" codeBase="http://www.sun.com/-">
      <permission class="java.io.AllPermission">
      </permission>
    </policyItem>
    <policyException signedBy="CK" codeBase="http://www.sun.com/-">
      <permission class="java.io.FilePermission">
        <permissionName name="/home/-"/>
        <actions name="write execute"/>
      </permission>
    </policyItem>
  </addItems>
</localPolicy>
```

Figure 4.2: A sample policy definition in JSEF demonstrating the use of policy exceptions

Since the same semantics can be achieved with either subtractive permissions or policy exceptions, the question arises whether both concepts are necessary. To our experience the answer is yes. If used as intended, the combination of the two concepts can simplify configurations and enhance readability. In the next section, more features of JSEF are introduced which necessitate the coexistence of subtractive permissions and policy exceptions.

4.2 JSEF's policy concept

As already mentioned in **Section 3** Java neither supports the concept of groups nor the enforcement of system-wide security settings. To overcome this drawback the policy concept of JSEF distinguishes between a *global* (defined by the administrator) and a *local policy* (defined by the user) which both can hold additive and subtractive permissions as well as policy exceptions. A user's local policy settings are under full control of the user and allow a user to define whatever privileges or restrictions he/she wants (with certain restrictions as will be described below). All examples presented so far have been taken from a user's local policy settings. The global settings on the other hand can only be modified by an administrator and apply to a group of or even all users. The scope of any global policy setting is determined by the user group it is assigned to. The concept of a *user group* is central to the definition of global policy settings. A user group is a collection of users who share a common set of permissions and restrictions which have been assigned to the group. To keep user groups maintainable, they are arranged in a hierarchical structure which facilitates inheritance of configurations (see **Figure 4.3**).

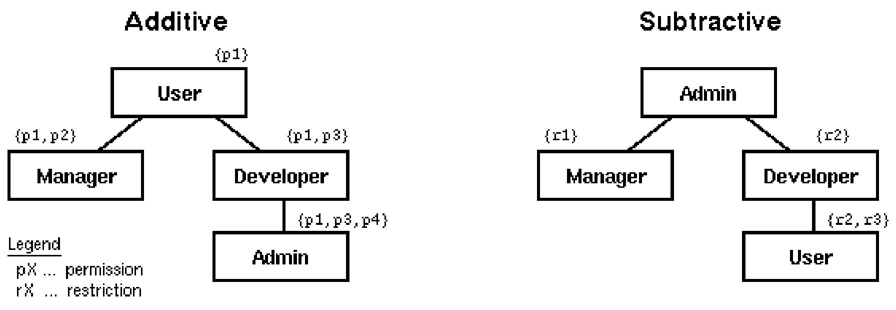


Figure 4.3: Hierarchical user groups

JSEF distinguishes between additive and subtractive hierarchies of user groups. In an additive hierarchy, permissions are broadened whereas in subtractive hierarchies the restrictions increase along the inheritance hierarchy (see **Figure 4.3**). A user group is assigned a set of additive or subtractive permissions and inherits all the settings of its parent groups. Hence, inheriting in this context means to collect all the permissions and restrictions of all parent groups. In **Figure 4.3** user group `Developer` inherits permission `p1` from group `User` and additionally defines `p3`. In the subtractive hierarchy group `Developer` does not inherit any restriction from group `Admin` but defines the restriction `r2`.

Although JSEF strictly separates additive and subtractive groups and their hierarchies to support clear and intuitive configurations, the power and applicability of this concept is not constrained. Additive subgroups always extend the permissions defined in their parent group while subtractive subgroups further restrict them. In the current implementation each group can have exactly one parent group of the respective (additive or subtractive) type. Although it would easily be possible to support multiple parent groups, the simplicity and clarity of the concept would be lost.

Every user can be member of one or several user groups and gains all the settings of all the groups he/she is member of. To enforce system-wide security policies, globally defined security settings always overrule local ones in the case of conflicting semantics. Thus, users cannot break the security policy defined by the administrator. They can, however, extend it in their local policy settings within the limitations of the global policy. This means that general security policies can be defined on a system-wide or even enterprise-wide level but still remain flexible enough to satisfy the needs of different users. E.g., enterprise-wide security settings could be stored on a web server and accessed using an SSL-HTTP connection whereas the local settings would be stored in XML files on a user's machine.

The distinction between local and global policies also motivates the need for both **policy exceptions** and **subtractive permissions**. Excluding privileges using policy exceptions in a global policy definition allows users to individually grant the excluded settings. A globally forbidden action, however, cannot be overruled by the user. From the user point of view, local policy exceptions simply exclude access to some resources when using wildcards. The use of subtractive local settings, on the other hand, allows a user to forbid actions even if they are granted by the global policy settings.
The format for defining permissions is given in **[14]**. **Figure 4.4** shows the assignment of user groups to a user (*Charly Brown*) who gets the permissions defined for group *User* and has the restrictions defined for group *Developer*. Since the *Developer* group is a subgroup of the *Admin* group (see **Figure 4.3**), the user is restricted by the subtractive permissions defined in both groups.

```
<?xml version="1.0"?>
<!DOCTYPE usergroups SYSTEM "usergroups.dtd">
<usergroups lastChanged="10/21/1999" changedBy="sysadmin">
  <user username = "Charly Brown">
    <addgroups>
      <group groupname = "User" />
    </addgroups>
    <subgroups>
      <group groupname = "Developer" />
    </subgroups>
  </user>
</usergroups>
```

Figure 4.4: Mapping of a user to groups in JSEF (group policy)

Figure 4.5 shows the definitions for subtractive user groups as depicted in

```
<?xml version="1.0"?>
<!DOCTYPE globalPolicy SYSTEM "globalPolicy.dtd">
<globalPolicy lastChanged="10/21/1999" changedBy="sysadmin">
  <group groupName="Admin">
  </group>
  <group groupName="Developer" parentGroup="Admin">
    <policyItem>
      <permission class="java.io.FilePermission">
        <permissionName name="/system/-"/>
        <actions name="read write execute delete"/>
      </permission>
    </policyItem>
  </group>
  <group groupName="User" parentGroup="Developer">
    <policyItem>
      <permission class="java.net.SocketPermission">
        <permissionName name="*" />
        <actions name="accept connect listen resolve"/>
      </permission>
    </policyItem>
    <policyException>
      <permission class="java.net.SocketPermission">
        <permissionName name="www.sun.com:8080" />
        <actions name="accept connect listen resolve"/>
      </permission>
    </policyException>
  </group>
</globalPolicy>
```

Figure 4.5: Definition of subtractive user groups in JSEF

The *Admin* group has no restrictions. Since the *Developer* usergroup is a subgroup of the *Admin* group which has no restrictions, it is restricted only by its own subtractive permissions which deny any access to the system directory. The *User* group is a subgroup of the *Developer* group, inherits its subtractive permissions and specifies a restriction which forbids all network connections independent of the destination host and the port. An exception to this restriction excludes the host *www.sun.com* at port 8080 from this rule (i.e., connections to *www.sun.com:8080* are not forbidden). Although the connecting to this host at the 8080 port is not explicitly forbidden by the subtractive policy settings, it additionally has to be allowed in the additive policy of a user to actually grant the permission. The user is free to define this permission in his/her local policy if he/she wants to permit the connection to *www.sun.com:8080*.

4.3 Handler Classes

Policy information such as user group definitions, local (user-defined) policies, and system-wide settings can be stored at any location using arbitrary protocols and formats to access them. JSEF provides interfaces to easily and dynamically customize the access to local or global settings. The high degree of flexibility of this concept is achieved by exploiting the *Abstract Factory* and *Factory Method* patterns as presented in **[4]**. **Figure 4.6** shows the UML class diagram for the local and global policy settings as used in our reference implementation which uses URLs to access configurations which are stored as XML documents.

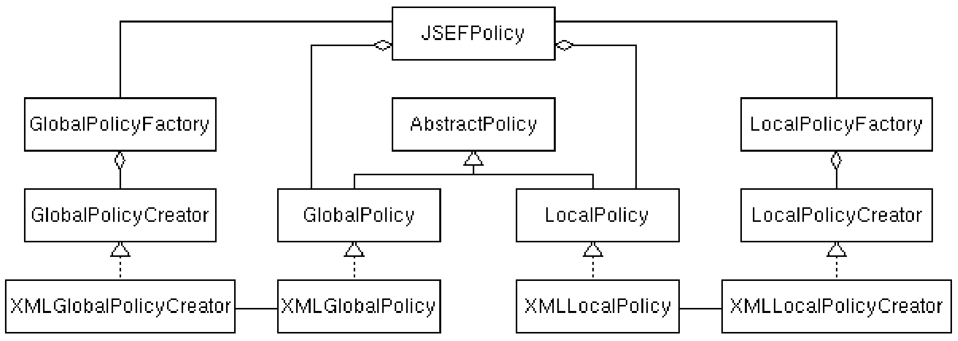


Figure 4.6: UML class diagram for the policy handler classes

New handler classes for other protocols or storage formats can easily be added to JSEF by simply adding a configuration option as described in **Section 6.3**.

4.4 Interactive policy negotiation

Once a user's policy has been constructed by merging the local and global policy settings, a class can be executed. As in Java's default model, any accesses to system resources are monitored by a security manager during runtime. Since JSEF includes concepts that extend the standard Java security policy, a specialized JSEF security manager is used. The JSEF security manager is implemented as a subclass of the standard security manager. As mentioned in **Section 2**, a security violation in Java's security model normally results in an abnormal termination of the Java virtual machine. If this occurs, a user would have to manually adapt his/her security settings and restart the application. This has to be repeated until all (unknown) permissions required by an application have been granted. In JSEF an interactive policy negotiation component takes care of this tedious problem and handles security violations during runtime. A security violation in the context of JSEF is an attempt to access a system resource which is denied due to one of three reasons:

1. The access was forbidden by a system-wide setting.
2. The access was forbidden by a user-defined setting.
3. The access was neither granted by system-wide nor by user-defined settings.

In the first case a user must not overrule the security decision since system-wide, subtractive settings cannot be influenced by the user and thus the application will not work. If, however, access to a resource is forbidden locally or the appropriate additive permission is not present in the policy settings, policy negotiation is started. In the current implementation of JSEF the user is questioned how to proceed, but basically any automatic or semi-automatic decision process could be used. If an additive permission is missing, the user can grant the appropriate permission. If access is forbidden locally, the user can add the appropriate permission to the **policy exceptions** of that configuration, thus allowing the action. In fact, such a permission or policy exception has to be added to all classes on the call stack which currently do not allow the requested action.

Whenever a security violation of type 2 or 3 is detected by JSEF's security manager, the user is informed *before* the action is carried out and, thus, he/she can influence the execution of the application in one of the following ways:

Do not allow action
The security exception is forwarded and the application usually is aborted.
Allow action once
The requested action is only allowed once. The next time the same operation is performed the user will be consulted again.
Allow action during the current session
The application is allowed to perform the requested action any number of times during the current JSEF session, i.e., during the runtime of the current JSEF instance.
Allow action permanently
An action is granted permanently which includes the previous option and additionally the permanent user-defined policy settings are updated accordingly.

The advantage of this concept is that a user can dynamically adapt his/her policy settings during runtime in a safe way. As a consequence, the current application need not be aborted any more.

4.5 The JSEF Process

On the basis of the previous discussion **Figure 4.7** summarizes the main steps of the process every piece of mobile code has to run through until it may be executed. This process is an extended but compatible version of Java's process as described in **Section 2**.

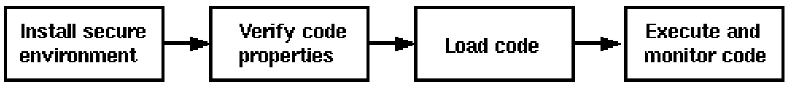


Figure 4.7: The JSEF Process

After a piece of code (a JAR file or a single class) has been downloaded, a secure environment in which this code will execute must be set up as a first step. This environment consists of the user-defined policy, the global policy defined by the administrator in the usergroups, and a security manager that cooperates with an access controller to enforce the security policy. This means that the actual permissions the mobile code will have at runtime must be determined in this step. For this purpose, the process merges the security policy defined by the user with the general security guidelines set up by the administrator as described in **Section 4.2**. The result of this process is a *policy object* that represents the combined additive and subtractive policy settings of the user-defined and the system-wide policies. The merging process is described in detail in [9] and [14]. This policy is installed and will be enforced by the JSEF security manager. It is instantiated and installed for the mobile code execution environment.

At this point a secure execution environment exists and the second step of the process can start: The properties of the mobile code (codebase and optional signature) must be checked and verified. The code's codebase, its signer, and the result of the signature verification determine its access permissions. The code's JAR file manifest must include the signer of the JAR file and the issuer of the signer's certificate. With this information the origin of the code can be proven and the signature of the JAR file can be verified (this is required by Java's class loading mechanism [7]). If JSEF is not used in interactive mode as described in **Section 6.1**, the manifest also explicitly must name the "main" class that is to be used to start the code in the JAR file.

Their properties having been proven, the classes in the JAR file can be loaded in step three. The loading is performed by a special JSEF class loader that extends the capabilities of the standard Java class loader [7] with enhanced functionalities such as loading classes from arbitrary locations. Finally, in step four, the mobile code can be executed by either calling the standardized start method of the code's "main" class or by letting the user choose interactively which start method to use (see **Section 6.1**). Again, this depends on whether JSEF is executed in interactive mode. During its execution, the mobile code will be monitored by the JSEF security manager and access controller that were set up for the execution environment in step one. Every access to resources will be intercepted and checked, possibly resulting in an interactive policy negotiation as described in **Section 4.4**.

5 JSEF Implementation - the gory Details

5.1 JSEF - Java Integration

In Java's default security architecture every attempt to access a system resource results in a call to a check method of the security manager. Which check method is called depends on the type of the access as well as the type of the resource. The check method of the security manager relays the decision to the access controller which in turn executes the **stack inspection algorithm** (see **Section 2**) to find out how to react. Since JSEF uses enhanced policy semantics as well as policy negotiation, the process of handling access requests had to be extended as well. **Figure 5.1** shows the adapted processing of an access request.

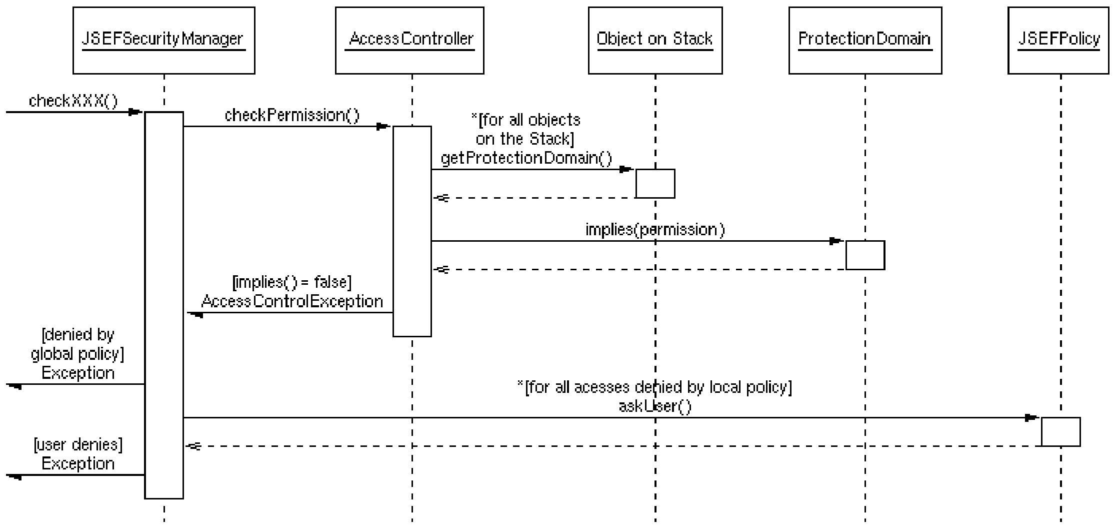

JSEFSecurityManager
AccessController
Object on Stack
ProtectionDomain
JSEFPolicy
checkXXX()
checkPermission()
*[for all objects on the Stack] getProtectionDomain()
implies(permission)
[implies() = false] AccessControlException
[denied by global policy] Exception
*[for all acesses denied by local policy] askUser()
[user denies] Exception

Figure 5.1: Processing of an access request in JSEF (UML sequence diagram [23])

Whenever a check method of JSEF's security manager is invoked by any of the Java core classes, it asks Java's access controller to check the appropriate permission. The access controller in turn iteratively requests the protection domain of every object on the stack (**stack inspection**) and tests every protection domain whether the access is allowed. If this check is positive the stack inspection continues. If not, an exception is raised. If the stack inspection terminates silently, i.e., no exception was raised, access to the requested resource is granted and the security manager returns silently, too. If, however, the access controller determined a class on the stack which is not granted the appropriate permission, an access control exception is thrown to the JSEF security manager. JSEF's security manager in turn first checks whether the access denial was due to a subtractive rule in the global policy. If yes, the access cannot be allowed, and an exception is raised to the executing code. If a local policy rule or a missing permission was the reason for the denial, the security manager has to figure out which classes on the stack lack the required permission by iteratively testing the protection domains of the classes on the stack. At this point the security manager has a list of classes lacking the requested permission or having locally forbidden it. Based on this list the user (or some other decision-making component) will be asked to decide whether to grant or deny the required permission. If the user denies, an exception is raised; otherwise the access can be performed.

The two key concerns in JSEF's security model implementation are how the security manager handles access requests and how the access controller applies JSEF's enhanced policy semantics. These two issues are discussed in the following.

5.1.1 JSEF's security manager

JSEF extends the default implementation of the security manager's check methods by adding support for policy negotiation as shown in **Figure 5.1**. **Figure 5.2** shows pseudo-code to explain how this is achieved.

```
public void checkXXX(parameters) {
  try {
    super.checkXXX(parameters);
  } catch (SecurityException se) {
    // start policy negotiation
    Permission perm = new ... // depending on the check method
    if (!negotiatePolicy(perm)) {
      throw new SecurityException("Not allowed: " + perm + "!\n" + se);
    }
  }
}
```

Figure 5.2: The concept of an extended check method in JSEF's security manager

Each check method in JSEF's security manager first forwards the request to the access controller by calling the corresponding method in the Java security manager (the superclass of the JSEF security manager). If the stack inspection of the access controllers throws a security exception, the JSEF security manager catches the exception and starts the policy negotiation. Depending on the check method that was called, the necessary permission is constructed and passed to the policy negotiating method. If the decision-making process (i.e., the user or some non-interactive component) does not grant the requested permission, a security exception is thrown to the caller. Otherwise, the method returns silently.

5.1.2 Enhanced policy semantics and the access controller

The access controller queries the protection domains of all classes on the call stack whether they grant the requested access as depicted in **Figure 5.1**. This means that the permission collection stored in the protection domain of the class is checked whether it implies the requested permission. In the case of JSEF this collection is a specialized `JSEFPermissionCollection` object that knows how to deal with JSEF's policy concept (additive and subtractive permissions, local and global policy, user groups). This special permission collection object is associated with the protection domain of the class when the class is being defined by the JSEF class loader (see **Section 2**). **Figure 5.3** shows the extended `implies` method of the `JSEFPermissionCollection` class. Its task is to decide whether a requested permission is included in the set of permissions stored in its protection domain.

```
public boolean implies(Permission p) {
  if ((p is globally forbidden) &&
      (p is NOT contained in a
         global subtractive exception)) {
    return false; // globally forbidden
  }
  if ((p is locally forbidden) &&
      (p is NOT contained in a
         local subtractive exception)) {
    return false; // locally forbidden
  }
  if ((p is globally allowed) &&
      (p is NOT contained in
         global additive exception)) {
    return true; // globally allowed
  }
  if ((p is locally allowed) &&
      (p is NOT contained in
         local additive exception)) {
    return true; // locally allowed
  }

  // neither forbidden nor allowed ->
  // not allowed!
  return false;
}
```

Figure 5.3: The extended `implies` method of the `JSEFPermissionCollection` class

Every `JSEFPermissionCollection` objects stores a set of global additive, global subtractive, local additive, and local subtractive permissions. Thus, the `implies` method has to check each of these four sets to determine whether a given permission is implied. Since each set consists of both permissions and policy exceptions, a given permission must be included in the permissions but not in the exceptions to be implied. Additionally, it is important to keep in mind that in JSEF subtractive permissions overrule additive ones which is why the subtractive permissions are checked first.

In-depth descriptions of the JSEF policy concepts are given in **[9]** and **[14]**. The processing of access requests is presented in detail in **[14]**.

5.2 Installing a customized policy class

As was pointed out in **Section 4**, JSEF replaces the system policy with its own enhanced policy object. Although the `policy.provider` setting that specifies the policy class to be used is defined in the Java system security properties file, a bug in the JDK 1.2 implementation prevents this from working correctly. In fact, the default policy based on policy files is always loaded regardless whether policy.provider is set to a different provider. In the current implementation of JSEF we circumvent this bug by starting JSEF with the `-Xbootclasspath` command line option which prevents loading the default policy files. After that, the JSEF policy which is a subclass of the `Policy` class can be installed using the `Policy.setPolicy` method.

The release notes of JDK 1.3 beta state that a customized policy class must be added to the `rt.jar` file containing the JDK classes. Only then the policy can be loaded by the bootstrap class loader. This has nearly the same effect as we achieve with our solution. The only difference is that with JDK 1.3's approach only the customized policy class will be loaded by the bootstrap class loader. Our approach using `-Xbootclasspath:<directory>` results in all files in the directory called `<directory>` to be loaded by the boot class loader. As a consequence, not only the customized policy class but all JSEF classes are trusted as if they were Java core classes.

5.3 Application of XML in JSEF

In JSEF all storage related tasks are based on interfaces. Support for XML is a matter of implementing these interfaces. If XML is used as the configuration format (as in the current implementation), an XML parser must be used for parsing the XML documents. Thus an `XMLParser` interface was defined which allows us to support different XML parsers. Currently the parsers of SUN Microsystems, IBM, and OpenXML are supported by JSEF. Generally, JSEF requires *valid* XML documents as input, based on the document type definitions (DTDs) presented in the **appendix**. After having parsed such an XML document, the policy information is transformed into an internal data representation depending on the kind of information. **Figure 5.4** shows the UML class diagram of JSEF's internal policy data structure.

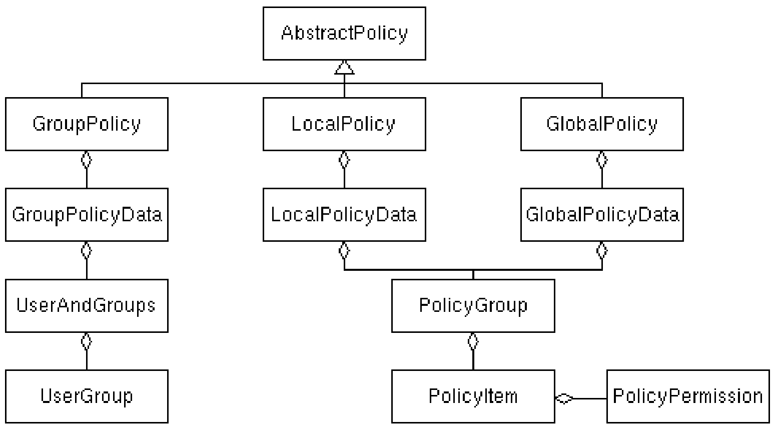


Figure 5.4: The internal data structure holding JSEF's policy information

`GroupPolicy` reflects the assignment of user groups to users as defined by the administrator. The group policy's data consists of `UserAndGroups` objects representing the users with their associated user groups (both additive and subtractive). Local and global policy settings are represented as `LocalPolicyData` and `GlobalPolicyData` classes. The local policy data consists of only two `PolicyGroup` objects, one for the additive, the other for the subtractive local policy. The global policy data, on the other hand, stores a set of policy groups representing the hierarchy of user groups and their privileges as presented in **Figure 4.5** . JSEF uses two global policy objects to keep track of the additive and subtractive global policy settings, respectively.

5.4 JSEF's Constraints

As was pointed out above JSEF's security manager overrides the check methods of the default implementation to integrate its enhanced policy concept. If, however, the `checkMemberAccess` method is overridden, the super class's implementation cannot be called any more. This constraint is due to the implementation of Java's security manager class which relies on a special stack depth of the code being checked. As a consequence, we did not override this method in the current implementation although it could be achieved with very little effort by re-implementing the method.

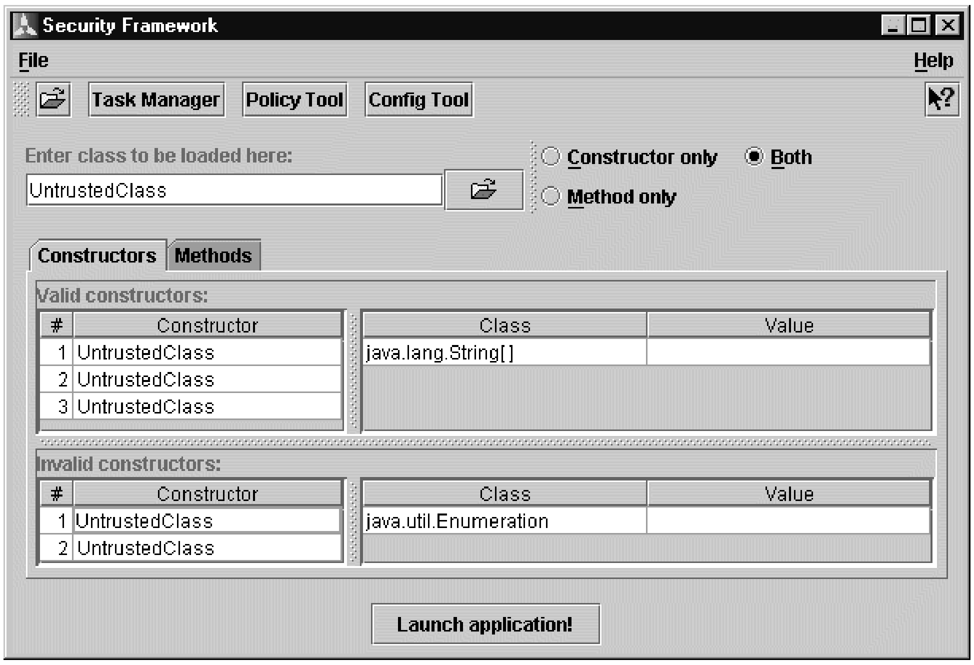
Another constraint concerns the interactive policy negotiation. In **Section 2** we discussed Java's stack inspection algorithm including the privileged execution mode. In the discussion of JSEF's policy negotiation feature in **Section 4.4** it was stated that missing privileges have to be added to all classes on the stack currently lacking them. If a privileged class is on the stack, however, it would suffice to add a missing privilege to only those classes on the stack which both are lacking the privilege and were called by the privileged class. Thus, the privileges of all classes which are 'shielded' from stack inspection by the privileged class would not need to be altered. Since JSEF cannot figure out which classes on the stack run in privileged mode, missing privileges are added to all classes on the stack instead of only those called by a privileged class.

Finally, a formal language for more sophisticated security policies can be imagined which might support boolean operators or an even more flexible constraint language. The usefulness and applicability of such a concept are to be investigated.

6 JSEF Tools

6.1 Secure Application Launcher

The *Secure Application Launcher* is the main front-end of JSEF and allows the user to execute classes inside the JSEF environment. In the default model, classes must fulfill special requirements such as having a `main` method to be executed. JSEF's secure application launcher (SAL) takes a more flexible and versatile approach on the basis of Java's Reflection API: SAL examines the given class and allows the user to choose any public method and constructor as a start method. **Figure 6.1** presents a typical scenario by showing the public constructors of the class `UntrustedClass`. A similar panel for its public methods is available (not shown in **Figure 6.1**).



Enter class to be loaded here: UntrustedClass

( ) Constructor only [✓ Both]
( ) Method only

Valid constructors:

#	Constructor
1	UntrustedClass
2	UntrustedClass
3	UntrustedClass

Class	Value
java.lang.String[]	

Invalid constructors:

#	Constructor
1	UntrustedClass
2	UntrustedClass

Class	Value
java.util.Enumeration	

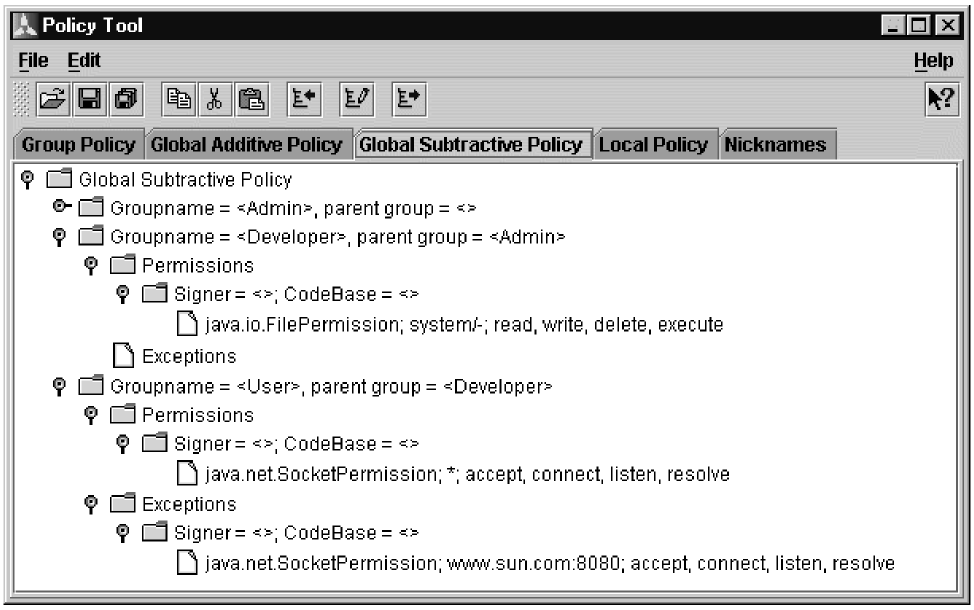
Figure 6.1: JSEF's secure application launcher

JSEF provides three ways of executing a given class. First, a method can be invoked directly if it is declared static. Second, a public constructor can be used to instantiate the given class and, third, a public constructor together with a non-static method can be invoked to instantiate a class and execute a given method. The user can choose one of these options and select a constructor and/or a method to be used. Since both the constructor and the selected method might require parameters, the SAL allows the user to specify values for those parameters. So far, the common parameter types such as integers, strings, or arrays of strings are currently supported by JSEF.

Additionally, SAL provides shortcuts to the **Policy Tool**, the **Configuration Tool** , and the task manager which lists all the tasks that currently use JSEF. A context-sensitive help facility (based on JavaHelp) supports the user in quickly understanding SAL.

6.2 Policy Tool

The *Policy Tool* can manage all policy related settings such as policy groups, local and global policy settings, and nicknames. It provides comfortable ways to edit the policy settings, context menus, context-sensitive help, and support for copy and paste of policy subtrees and the contained settings. **Figure 6.2** shows an example of a global subtractive policy configuration including the hierarchy of user groups, the permissions, and the defined policy exceptions.



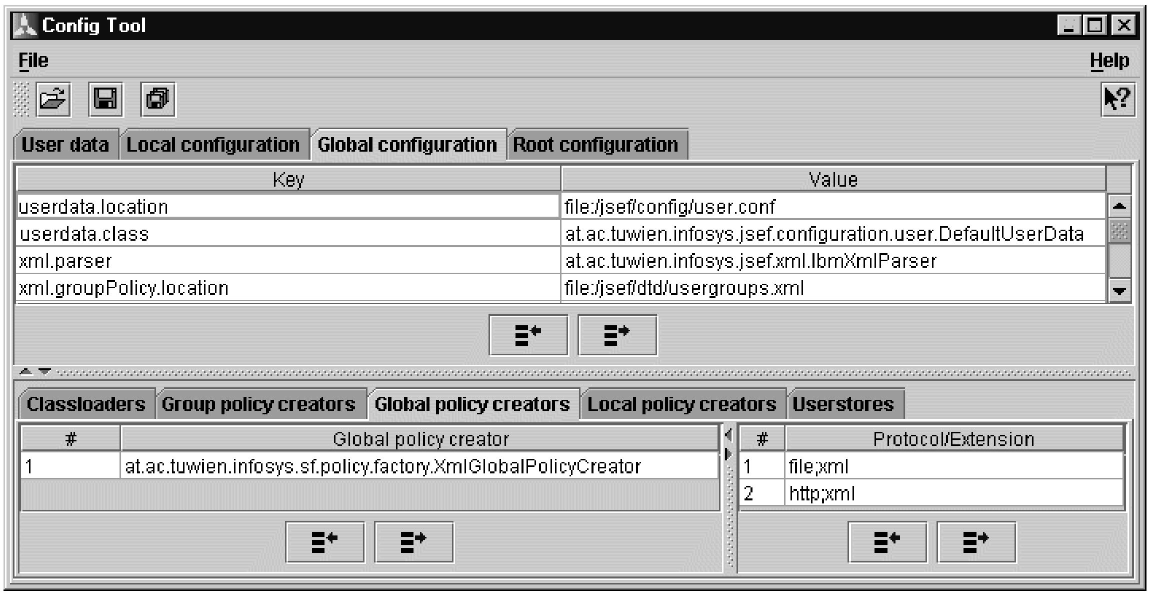
```
Global Subtractive Policy
  Groupname = <Admin>, parent group = <>
  Groupname = <Developer>, parent group = <Admin>
    Permissions
      Signer = <>; CodeBase = <>
        java.io.FilePermission; system/-; read, write, delete, execute
    Exceptions
  Groupname = <User>, parent group = <Developer>
    Permissions
      Signer = <>; CodeBase = <>
        java.net.SocketPermission; *; accept, connect, listen, resolve
    Exceptions
      Signer = <>; CodeBase = <>
        java.net.SocketPermission; www.sun.com:8080; accept, connect, listen, resolve
```

Figure 6.2: JSEF's Policy Tool

By default JSEF uses XML documents to represent all policy settings. Thus the hierarchy shown in **Figure 6.2** reflects the hierarchical structure of the corresponding XML documents as defined in JSEF's DTDs (see **appendix**). **Figure 6.2** only shows a small part of the available functionality. With the policy tool the user can additionally define group policies, the global additive policy, the local policy, and manage nicknames. Nicknames allow the user to assign simple string names to distinguished names which are used to identify certificates but are complicated to remember.

6.3 Configuration Tool

The *Configuration Tool* allows the user to configure JSEF itself (**Figure 6.3**).



Key	Value
userdata.location	file:/jsef/config/user.conf
userdata.class	at.ac.tuwien.infosys.jsef.configuration.user.DefaultUserData
xml.parser	at.ac.tuwien.infosys.jsef.xml.IbmXmlParser
xml.groupPolicy.location	file:/jsef/dtd/usergroups.xml

#	Global policy creator
1	at.ac.tuwien.infosys.sf.policy.factory.XmlGlobalPolicyCreator

#	Protocol/Extension
1	file;xml
2	http;xml

Figure 6.3: JSEF's Configuration Tool

The following list gives a short overview of the available options:

User data
The user data view includes the user management front-end of JSEF. Users can be added or removed and their privileges and the location of their local policy definitions can be defined. The privileges of a user include whether the user is allowed to change the various configuration settings and whether he/she may alter the local or global policy settings.
Local and global configuration
These settings include a set of general purpose configurations. They define which XML parser is used, the locations of the global additive and subtractive policy definitions, the location of the help information, and which class shall be used to store user information among others.
Furthermore, multiple class loaders, handler classes with their associated protocols and extensions, and a number of repositories for user authentication can be defined here.
Root configuration
The root configuration view comprises only a single setting which, however, is crucial to the functioning of JSEF: The root configuration setting defines where the global policy definitions of a user can be found. Obviously, it is important to ensure that the root configuration setting can only be changed by the system administrator.

7 Related Work

While the Java security model has been the subject of active discussion since the introduction of Java, the administration of security policies and profiles has received little attention. Java's security model is well-documented [5, 6, 7] and many approaches exist to extend or replace this basic model in terms of new security features and capabilities. For example, [16] describes an approach which uses protected domains, so-called playgrounds, to protect machines and resources from mobile code. A playground is a dedicated machine on which the mobile code is executed, with its input and output re-directed to the user's machine. This creates the illusion that the mobile code is executed on the user's computer while it is actually run on the playground machine which is physically separated from the user's machine and thus the mobile code has no access to the user's resources.

The J-Kernel [12] goes even further by replacing the standard Java security architecture with a capability-based system that supports multiple cooperating protection domains inside a single Java virtual machine. While Java's protection domains are closer to the notion of a user, J-Kernel defines them more like processes which considerably changes the security semantics. Via protection domains J-Kernel separates objects into local ones and capability objects which are shared among domains. It provides capability-based communication channels and supports revocation of capabilities.

The security model for aglets **[13]** uses concepts closely related to JSEF but targets the specific needs of mobile agents. Aglets are mobile agents which execute in a certain context on any aglet-aware host they visit. The aglet security model defines the concept of principals to separate the security requirements of the owner, the manufacturer, and the context master of an aglet. The principals define layers of security in which security settings can be refined but not overruled. If an aglet matches several policy definitions, a "consensus voting rule" combines the policy settings. Since aglets are mobile agents the privileges which define access to local resources are augmented with privileges defining inter-aglet behavior and allowances. Allowances are privileges encapsulating system resources such as memory usage and CPU time, whose implementation and enforcement require (incompatible) modifications to the Java virtual machine. Similar to JSEF, users may be grouped in named groups and share a set of permissions. Policy definitions may be combined with simple boolean operators which supports composite privileges and negation of privileges. Furthermore, black-lists exist to disallow suspicious aglets and contexts. JSEF's subtractive permissions and policy exceptions can be expressed in the aglet model by the use of boolean operators. The different principals imposing policy restrictions on an aglet relate to local and global policies in JSEF. While the aglet security model extends the Java security model to provide inter-aglet permissions, JSEF only builds on the permissions defined by Java. In contrast to the aglet model, JSEF facilitates to structure user groups hierarchically which supports simpler and less error-prone administration of security profiles.

An interesting conceptual approach to extend Java's security features and simplify definition of security profiles is presented in **[20]**. This approach suggests a constraint language which allows the user to specify security constraints which are a combination of subject-based, object-based, and history-based policy statements. History-based constraints are a powerful concept and support the definition of policy rules over time. For example, it could be specified that an applet can only make 5 write accesses to a file. Additionally rules can specify conditional constraints, such as if a piece of mobile code wishes to access a protected file it no longer can make a network connection. Constraints can be combined with simple logical operators and can define both additive and subtractive permissions (similar to JSEF). Policy exceptions are not an explicit concept but can be defined implicitly. No grouping mechanisms and hierarchies exist which makes it difficult to assign layered security profiles to users. The definition of constraints is cumbersome since an S-expression type language is used and no graphical tool support is available. This approach has been applied to and tested only with JDK 1.1. However, we plan to further investigate the addition of a constraint language as suggested in **[20]** to JSEF to further extend JSEF's flexibility and configuration options.

In contrast to these approaches, JSEF does not introduce incompatible Java security features. Instead it uses the existing Java security architecture but enhances its usability, introduces higher-level abstractions and hierarchical policies, and offers new ways of configuration as described in the previous sections. It simplifies the definition and maintenance of security policies at the system and at the user levels. Such simplification facilitates to prevent the introduction of security holes and thus improves a system's overall security characteristics. None of the above approaches supports (interactive) runtime security negotiation and offers simple ways to define system-wide (or even network-wide) security policies as JSEF. Also tool support for security maintenance is very limited or does not exist at all.

8 Conclusion

The Java Secure Execution Framework (JSEF) presented in this paper was built on top of Java's standard security architecture and extends it with powerful features in a compatible way. JSEF provides a security framework which simplifies the maintenance of security profiles, provides graphical tool support for security administration, and adds useful yet lacking security features to the standard Java security model.

While in standard Java only permitted accesses can be defined, which can blow up configurations and makes them cumbersome to maintain, JSEF additionally supports the specification of forbidden accesses (subtractive policy) and policy exceptions which simplifies security settings and their comprehension. JSEF's hierarchical groups provide a concept to aggregate users into groups, freely structure these groups into a hierarchy, and assign security profiles to them. This supports simpler maintenance and tailoring of security policies according to users' needs. The concept of global and local policies in JSEF enables the definition of network-wide security policies that define a security corset for users while still allowing them to freely adjust their configurations inside these mandatory security standards. Thus users can tailor this policy towards their needs but cannot break it. By default the global policy always overrules the user-defined local policies.

Since in the standard Java security model, a forbidden access typically terminates the execution, JSEF offers the possibility to negotiate security interactively at runtime. This avoids tedious and cumbersome trial-and-error cycles to find out about the actual permissions required by a piece of mobile code, which is especially helpful for applets. It also allows the user to grant the required permissions exactly in a comfortable way. Without such a facility frequently far too high permissions are granted. JSEF intercepts forbidden accesses at runtime and the user (or a special security control component) can negotiate with the relevant Java code about the requested permissions.

For convenient use and configuration JSEF offers powerful and flexible graphical user interfaces which support the user or administrator in the definition and maintenance of security relevant issues such as profiles and user groups. JSEF is intended to improve and simplify the administration of security rather than it is a technique for supplying higher security to mobile code environments. However, better maintenance support also dramatically improves overall system security since it helps to prevent sloppy configurations or the introduction of security holes by erroneous configurations.

References

[1] T. Bray, J. Paoli, and C. M. Sperberg. Extensible Markup Language (XML) 1.0. World Wide Web Consortium (W3C), 10 February 1998. W3C Recommendation. **http://www.w3.org/TR/1998/REC-xml-19980210.html**.
[2] G. Coulouris, J. Dollimore, and T. Kindberg. Security. In *Distributed systems - concepts and design*, International Computer Science Series, pages 477-516, 2nd edition. Addison-Wesley, Reading, Mass. and London, 1994.
[3] S. Fritzinger and M. Mueller. Java security. Sun Microsystems, Incorporated, 1996. White Paper. **http://java.sun.com/security/whitepaper.txt**.
[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*, Addison-Wesley, Harlow, England, October 1995.
[5] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: an overview of the new security features in the Java Development Kit 1.2. *Proceedings of the USENIX Symposium on Internet Technologies and Systems* (Monterey, California, December 1997). USENIX Association, 1997.
[6] L. Gong and R. Schemers. Implementing Protections Domains in the Java Development Kit 1.2. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, San Diego, California, March 1998. **http://www.javasoft.com/people/gong/papers/pubs98.html**.
[7] L. Gong. Secure Java Class Loading. *IEEE Internet Computing*, **2**(6):56-61, November/December 1998.
[8] S. Gritzalis and D. Spinellis. Addressing threats and security issues in world wide web technology. *Proceedings of CMS'97, 3rd IFIP TC6/TC11 International Joint Working Conference*

on Communications and Multimedia Security (Athens, Greece), pages 33-46, September 1997.
[9] M. Hauswirth, C. Kerer, and R. Kurmanowytsch. Minstrel Client Security Framework. Distributed Systems Group, Technical University of Vienna, Austria, 1999. **http://www.infosys.tuwien.ac.at/Minstrel/Receiver/CSF/**.
[10] M. Hauswirth and M. Jazayeri. A Component and Communication Model for Push Systems. *Proceedings of the ESEC/FSE 99 - Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)* (Toulouse, France, September 6--10, 1999), September 1999. **http://www.infosys.tuwien.ac.at/Staff/pooh/papers/PushIssues/**.
[11] M. Hauswirth, *Internet-Scale Push Systems for Information Distribution---Architecture, Components, and Communication*. PhD thesis. Distributed Systems Group, Technical University of Vienna, October 1999. **http://www.infosys.tuwien.ac.at/Staff/pooh/diss/Thesis.ps**
[12] C. Hawblitzel, C.-C. Chang, G. Czajkowski, D. Hu, and T. von Eicken. Implementing Multiple Protection Domains in Java. *Proceedings of the USENIX Annual Technical Conference* (New Orleans, Louisiana, June 1998). USENIX Association, 1998.
[13] G. Karjoth, D. B. Lange, and M. Oshima. A Security Model for Aglets. *IEEE Internet Computing*, **1**(4), July 1997.
[14] C. Kerer. *A flexible and extensible security framework for Java code*. Master's Thesis. Distributed Systems Group, Technical University of Vienna, Austria, October 1999. **http://www.infosys.tuwien.ac.at/Teaching/Finished/MastersTheses/JSEF/**
[15] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, Mass. and London, 1997.
[16] D. Malkhi, M. K. Reiter, and A. D. Rubin. Secure Execution of Java Applets using a Remote Playground. *Proceedings of the IEEE Symposium on Security and Privacy*, Los Alamitos, California, May, 1998. **http://www.cs.nyu.edu/~rubin/playground.ps**.
[17] G. McGraw and E. Felten. Java security and type safety. *Byte*, **22**(1):63-4, January 1997.
[18] G. McGraw and E. W. Felten. *Java security: hostile applets, holes, and antidotes*. John Wiley, New York, 1997.
[19] G. McGraw and E. W. Felten. *Securing Java: getting down to business with mobile code*. John Wiley, New York, 1999.
[20] N. V. Mehta and K. R. Sollins. Expanding and Extending the Security Features of Java. *Proceedings of the 7th USENIX Security Symposium* (San Antonio, Texas, January 26-29, 1998). USENIX Association, 1998.
[21] The Minstrel Push System Project website. Distributed Systems Group, Technical University of Vienna, 1999. **http://www.infosys.tuwien.ac.at/Minstrel/**.
[22] A. D. Rubin and D. E. Geer. Mobile Code Security. *IEEE Internet Computing*, **2**(6):30-4, November/December 1998.
[23] J. Rumbaugh, I. Jacobson, and G. Booch. *Unified Modeling Language Reference Manual*, Object Technology Series. Addison-Wesley, Reading, Mass. and London, 1999.
[24] Sun Microsystems, Incorporated. *Secure computing with Java: now and the future*, September 1998. White Paper. **http://java.sun.com/marketing/collateral/security.html**.
[25] F. Yellin. Low level security in Java. *Fourth International World Wide Web Conference* (Boston, Massachusetts, USA, December 11--14, 1995). Published as *World Wide Web Journal*, **1**(1). O'Reilly & Associates, Incorporated, November 1995. **http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html**.

Vitae

Manfred Hauswirth is an assistant professor at the Distributed Systems Group at the Technical University of Vienna, Austria. He holds an M.Sc. and a Ph.D. in computer science from TU Vienna. His research interests include push systems, mobile code, payment systems, World-wide Web, Internet/intranet systems and applications, programming languages, and distributed systems. After

several jobs in industry in the fields of distributed applications, portability, and databases he joined TU Vienna as a research assistant in 1994. He is the principal researcher in the Minstrel push system project and a senior researcher in the OPELIX and MOTION EU projects. He is a member of IEEE and ACM.
Clemens Kerer is a research assistant and Ph.D. student at the Distributed Systems Group at the Technical University of Vienna, Austria. He holds an M.Sc. in computer science from TU Vienna. His interests include distributed systems, Java security, World-wide Web, Intranet systems, software components, and patterns.
Roman Kurmanowytsch is a visiting researcher at the Hewlett-Packard Laboratories, Bristol, UK and a Ph.D. student at the Distributed Systems Group at the Technical University of Vienna, Austria. He holds an M.Sc. in computer science from TU Vienna. His interests include Voice over IP, Internet security, World-wide Web, and Java.

Appendix

Document Type Definitions

Group Policy DTD

```
<?xml encoding="US-ASCII"?>
<!ELEMENT usergroups (user)*>
<!ATTLIST usergroups lastChanged  CDATA  #REQUIRED
                     changedBy    CDATA  #REQUIRED>
<!ELEMENT user  (addgroups | subgroups)*>
<!ATTLIST user  userName ID  #REQUIRED>
<!ELEMENT addgroups (group)+>
<!ELEMENT subgroups (group)+>
<!ELEMENT group EMPTY>
<!ATTLIST group groupName  CDATA  #REQUIRED>
```

Local Policy DTD

```
<?xml encoding="US-ASCII"?>
<!ELEMENT localPolicy (addItems | subItems)*>
<!ATTLIST localPolicy userName     ID     #REQUIRED
                      lastChanged  CDATA  #REQUIRED>
<!ELEMENT addItems (policyItem | policyException)+>
<!ELEMENT subItems (policyItem | policyException)+>
<!ELEMENT policyItem (permission)*>
<!ATTLIST policyItem signedBy  CDATA  #IMPLIED
                     codeBase  CDATA  #IMPLIED>
<!ELEMENT policyException (permission)*>
<!ATTLIST policyException signedBy  CDATA  #IMPLIED
                                codeBase  CDATA  #IMPLIED>
<!ELEMENT permission (permissionName?,actions?)>
<!ATTLIST permission class  CDATA  #REQUIRED>
<!ELEMENT permissionName EMPTY>
<!ATTLIST permissionName name  CDATA  #REQUIRED>
<!ELEMENT actions EMPTY>
<!ATTLIST actions types NMTOKENS #REQUIRED>
```

Global Policy DTD

```
<?xml encoding="US-ASCII"?>
<!ELEMENT globalPolicy (group)*>
<!ATTLIST globalPolicy lastChanged  CDATA  #REQUIRED
                       changedBy    CDATA  #REQUIRED>
<!ELEMENT group (policyItem | policyException)*>
<!ATTLIST group groupName    ID     #REQUIRED
                      parentGroup  IDREF  #IMPLIED>
```

```
<!ELEMENT policyItem (permission)+>
<!ATTLIST policyItem signedBy  CDATA  #IMPLIED
                           codeBase  CDATA  #IMPLIED>
<!ELEMENT policyException (permission)+>
<!ATTLIST policyException signedBy  CDATA  #IMPLIED
                          codeBase  CDATA  #IMPLIED>
<!ELEMENT permission (permissionName?,actions?)>
<!ATTLIST permission class  CDATA  #REQUIRED>
<!ELEMENT permissionName EMPTY>
<!ATTLIST permissionName name  CDATA  #REQUIRED>
<!ELEMENT actions EMPTY>
<!ATTLIST actions types CDATA #REQUIRED>
```