An Access Controller for Java Card

[Published in *Proc. of Gemplus Developer Conference*, Paris, France, June 20–21, 2001.]

André Amegah, Laurent Gauteron, and Pierre Girard

Gemplus, Security Technology Department,
B.P. 100, 13881 Gémenos Cedex, France
{Laurent.Gauteron, Pierre.Girard}@gemplus.com

Abstract. Until now the Java Card security mechanisms don't include an access controller such as in the Java virtual machine. In this paper, we advocate that there is now a need for such a mechanism, and we present our implementation of a Java Card access controller.

1 Introduction

Java security mechanisms were originally designed against the threat of aggressive applets: code coming from the Internet is potentially dangerous and could, without adequate security mechanisms, attack the JVM, the underlying operating system, and finally all the data and the application of the JVM host.

Java security has evolved gradually through the successive releases of the JDK. The very first concept introduced by the Java security designers was the *sandbox*. This refers to the limitation of an applet's actions and was enforced by a *security manager* which was used to restrict API calls that can be made by an applet to the very few proved to be harmless.

To ensure that this first defence could not be bypassed with an aggressive piece of code, the JVM relies on the type safety of the Java language, and on the byte code verifier which verifies that each piece of code dowloaded in the JVM is in conformance with the Java specification (i.e. is type-safe).

In the JDK 1.1, the notion of *signed applets* was introduced. This allowed to overcome the all-or-nothing security of the *sandbox* paradigm. It was possible to specify which signers were trusted. At download time the applet digital signature was verified, and if recognised as trusted, the applet was treated as local code and granted full access to the system.

This model was significantly better than the original one, but still not very flexible as a fine-grained access control could only be achieved through a custom security manager coded in Java.

The JDK 1.2 introduced a new mechanism called the access controller which uses an external security policy stating fine-grained rights for applets based on their origin and their digital signature. In this new design, the security policy can be easily modified by editing a file. The security manager has been kept

for backward compatibility, and does nothing but passing calls to the access controller (see figure 1 for the summary picture of the Java mechanisms).

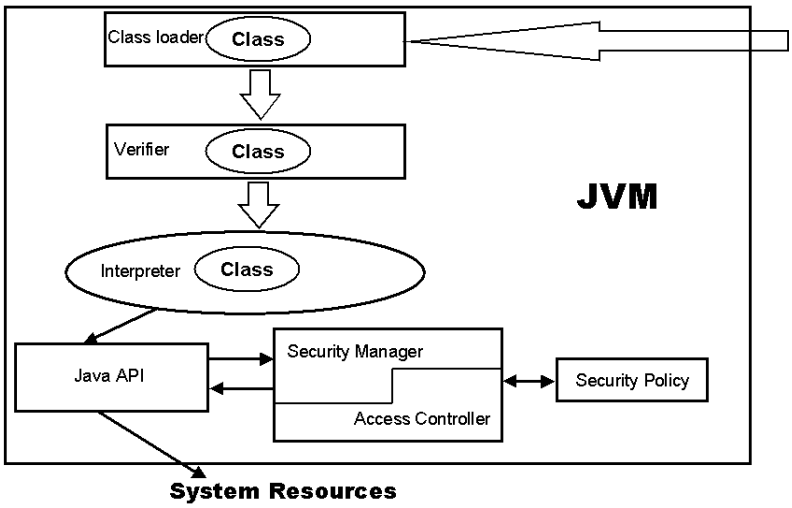

Fig. 1. The Java security mechanisms

The current Java Card security mechanisms are lighter and do not include the sandbox concept. However, Java Card benefits from the Java type safety and from the byte code verifier, which is usually offboard. Applet downloads are strictly controlled by a secure loader, generally the *de facto* standard is VOP [3]. Although the previous mechanisms are jointly able to prevent an aggressive applet to damage the JCVM, the underlying OS or the other applets, a second barrier has been built: the firewall. It allows to ensure that the card remains in a secure state even if the verifier is bypassed. The firewall is implemented by verifications at interpretation time to check that each memory access is correct, i.e. that an applet does not forge or use a pointer to access a memory location which does not belong to the applet. Until now, no security manager or access controller has been implemented. This was perfectly sound as the API was so restricted that no malicious action could be foreseen using them. For instance, there is no global file system in a Java Card, thus no file-related API and consequently no associated malicious file usage.

However, we think that the situation has evolved, and that the Java Card architecture could now benefit from an access controller. This paper presents the Java Card access controller we have designed and implemented. In a first section we give our arguments in favour of the introduction of an access controller

for Java card, then we recall how the Java access controller works. In a third part, we sketch the architecture of our Java Card access controller, and finally we present the implementation and directions to use it.

2 Motivations for the Java Card access controller

We mainly see three kinds of needs for a Java Card access controller: restriction of platform API usage, restriction of proprietary API usage and cryptography-related regulations enforcement.

Until now, the set of Java Card API was restricted enough to prevent its misuse. However this API evolves and will propose more and more services to applets, and it will become necessary to control precisely which applet can use which service. Even now, the VOP standard offers some services to the applets which should not be given to every one of them. For example, the card global PIN can be presented internally by an applet through the `verifyPIN` method. Unfortunately, an attack using this method to illegally recover the PIN code has already been published [1]. In addition, in some commercial schemes, it is mandatory to let some applets use the global PIN, whereas some others shouyld not, because of the existence of different agreements and pre-existing schemes. In short, filtering calls to the platform API is mainly considered for security reasons.

In a classical system, multiapplicative smart cards are usually managed by the card issuer which allows some application providers to download additional applets. The issuer can provide services to the additional applets through proprietary API. However, access to these services could be available for a fee or only to first class partners. This is especially true if the proprietary API provides a service that saves code in the caller applet or value-added information such as transaction records or information on the personal profile of the holder. In short, filtering calls to the proprietary API is mainly considered for business reasons.

Cryptographic services provided by Java Card API are widely-used by applets to build their security. Unfortunately, many countries around the world implement numerous regulations on cryptography import, export, and even domestic use. Generally these regulations do no prohibit the use of cryptographic primitives, but the building of encryption machines with them (see [2] for an extensive discussion on this subject). Generic products such as Java Card need to take this into account, and should authorise a well known applet (e.g. a given electronic purse) to use cryptographic primitives, but not unknown ones which can, for example, encrypt and decrypt arbitrary data. Consequently, card manufacturer, resellers, integrators and card issuer, will be able to take advantage of a Java Card access controller to enforce local regulations, and than, to deploy quickly and legally their smart card applications.

Before describing into details our Java Card access controller, we first recall in the next section the principles of the Java access controller.

3 The Java access controller

The foundations for access control are given by the security policy which allows to set precisely the rights of each class. Inside the JVM, the security policy is implemented by an object which can be queried (using the `getPermissions API`) to check the permissions granted to a piece of code. In the JDK reference implementation, the policy object refers to an external file containing the security policy rules set by the user.

The rules are expressed with two basic properties of the classes which are their origin (i.e. their URL or code source in the java terminology) and their digital signature. These two properties fully characterise a piece of code from a security policy stand point and are encapsulated in an object of the `CodeSource` class.

Logically, the argument of the `getPermissions API` is a `CodeSource` object and a call returns a set of permissions. Each permission object expresses a single precise right which is granted. Permission classes inherit from `java.security.Permission` and implement an `implies` method which expresses the semantics of the permission (i.e. give a boolean answer to know if an action is authorised or not by the permission). For example the file permission `p` is constructed as:

```
p = new FilePermission(‘‘/tmp/*’’, ‘‘read, write’’);
```

will imply the permission `q` such that:

```
q = new FilePermission(‘‘/tmp/myFile’’, ‘‘read’’);
```

Instead of associating directly the permissions to a class, the virtual machine uses an indirection through a protection domain. Protection domain objects keep the set of permission granted to a given code source and each class belongs to a protection domain. Classes having the same code source belong to the same protection domain. This approach gives a better flexibility and limits the number of permission objects.

The access controller itself cannot be instantiated and provides the static method `checkPermission`. This method takes a permission object and returns silently if the permission is granted to the class, otherwise, a `SecurityException` is thrown. So when a class `C` uses the API to read a file, the invoked `read` method instances a new permission object (like `q` in the previous example) and invokes `checkPermission(q)` of the security manager which, in turn, call `checkPermission(q)` of the access controller.

This summary of the Java access controller, although very short, gives enough information to understand its design rationale. In the next section we introduce the architecture of the Java Card access controller.

4 The architecture of the Java Card access controller (JCAC)

It is clearly impossible to simply transpose the Java access controller inside a smart card due to the card architecture, and, in particular its tiny memory. Even

though we keep the same concept of access control, it is strongly scaled down and adapted to the Java Card environment. In addition of the memory footprint, our architecture takes into account the fact that objects are persistent, that multiple applets can run in the same JCVM and that the only available communication mean is the APDU protocol.

The JCAC is an applet offering the `checkPermission` service to other applets and libraries which communicates through APDU with a client used to administrate the card security policy (see figure 2). The JCAC applet maintains some data structures reflecting the security policy although there is no security policy object.

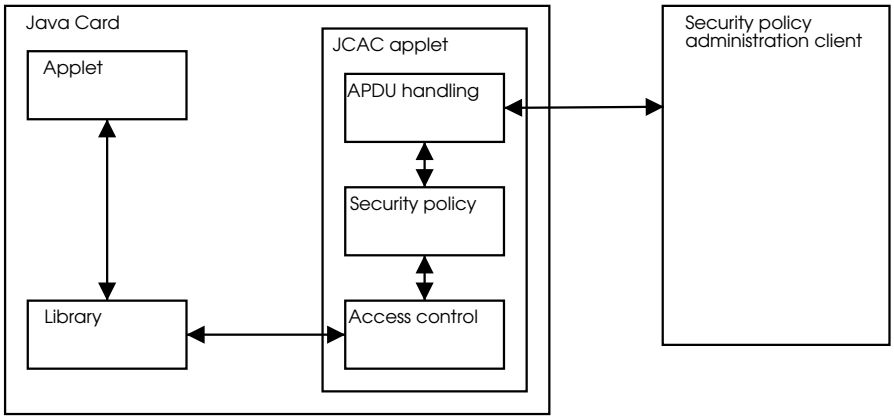

Fig. 2. The Java Card Access Controller

For that purpose, permissions are grouped in protection domains. Each applet is assigned to a unique protection domain. If the security policy gives the same rights to two applets, they are placed in the same protection domain. This approach differs from standard Java: the permission are granted to applets instead of classes which is less precise. However this is perfectly acceptable if one considers that Java applets can be built with classes from different origin with the class loader mechanism, whereas classes used in a Java Card applet have to be bundled and loaded with the applet in one package.

The code source of an applet is, at present time, simply represented by its AID. As applets are securely loaded (and therefore signed) we could expect to extend the definition of the code source to include the signer. Another possible refinement would be to use partial AID. In that case, it would possible to use a mechanism similar to the `implies` method of the `CodeSource` class to decide if a given AID fits in a protection domain associated with a partial AID.

The access control mechanism analyses the security policy when invoked by an API willing to check if it is legally called or not. This API passes as arguments

the permission to check along with the AID of its caller. The control in itself is fairly straightforward: the matching protection domain is retrieved and the permission to check is compared with all the granted permissions. As in the Java case, if the permission is granted, the method returns silently, otherwise, an exception is thrown.

5 The implementation of the Java Card access controller

Our JCAC applet has been implemented on a Gemplus GemXpresso 211 V2 smart card which is compliant with the Java Card 2.1 and VOP 2.0 standards.

5.1 Protection domain, permissions, and actions

In the current prototype stage, the protection domains and the applets are only related by AID and not by partial AID or applet signers. The permissions and the actions have not a name of type `String`, but `byte`. Each protection domain contains a list of permissions and each permission contains a list of actions.

However, we don't have a permission class and don't manipulate permissions as objects. Consequently we can't have an embedded semantic inside a permission (as it is the case in Java with the `implies` method).

This allows a simple argument passing through a shareable interface. Passing permissions as objects would have caused some problems to transfer them from an API context to the access controller context: the firewall would have prevented any access to the objects' content from the access controller context. This could be solved if the JCAC is integrated in the JCRE (in that case it could access everything) or if the JCRE creates instances of permissions as entry point objects.

We should mention that, as permissions and actions are represented by a `byte`, there is a need for a common and standardised interpretation of these constants. They should be allocated for system API needs, but also for other standard API (such as VOP). Finally a subset needs to be reserved for proprietary API.

5.2 The `checkPermission` method

First, a reference to the JCAC shareable interface needs to be obtained through an invocation of `JCSystem.getAppletShareableInterfaceObject(JCAC_AID, (byte) 0)`. Once again, the AID of the JCAC needs to be standardised in this prototype configuration. Ideally, the JCAC will be integrated in the JCRE, we can imagine that its reference could be retrieved with a method such as `JCSystem.getAccessController()`, or even better that the AccessController class could offer a static `checkPermission` method exactly like in Java. For security reason, the JCAC applet is final and can not be installed with another AID than the standard AID hard-coded in the applet.

For now, the JCAC applet offers an shareable interface with the `checkPermission` method:

```
public interface AccessControllerInterface  extends Shareable {
    public void checkPermission (AID aid, byte permission,
                                 byte action);
}
```

As mentioned before, the permission and the action are given as arguments of type `byte`. If the permission to check contains more than one action, the API will have to call `checkPermission` for each action.

The AID of the caller is passed just because our applet is not integrated in the system and cannot benefit from a stack introspection mechanism to identify the callers in the stack.

5.3 The security policy management

The security policy is managed from the outside thanks to the following APDUs:

- AddPermission: This APDU contains an action to add to a permission in a given protection domain. If not created yet it creates a new protection domain in the security policy and a new permission in that protection domain.
- RemoveAll: This command initialises the security policy and removes all the existing protection domains and permissions.
- GetFirstAction and GetNextAction: These commands allow a terminal to retrieve the security policy contained into a card to subsequently browse and modify it.

All these commands need to be performed in a secure messaging mode, i.e. using a VOP secure channel.

5.4 The client for policy management

If the terminal side has an image of the security policy contained in the card (for example trough a card management system), it can directly issue APDUs to maintain the security policy. However, a snap-shot of the policy can also be retrieved before issuing such commands.

For test and diagnostic purposes, we have developed a GUI client of the JCAC applet which allows an administrator to retrieve, browse and modify graphically the security policy. A screen shot is given in figure 3.

6 Conclusion

In this paper, after a presentation of the current Java mechanism, we have explained why an access controller should be useful on the Java Card platform. We have presented the architecture and design of a first prototype of such an access controller. As the size of our prototype is around 2 Ko, it can easily be introduced as a standard feature of Java Card at a small cost.

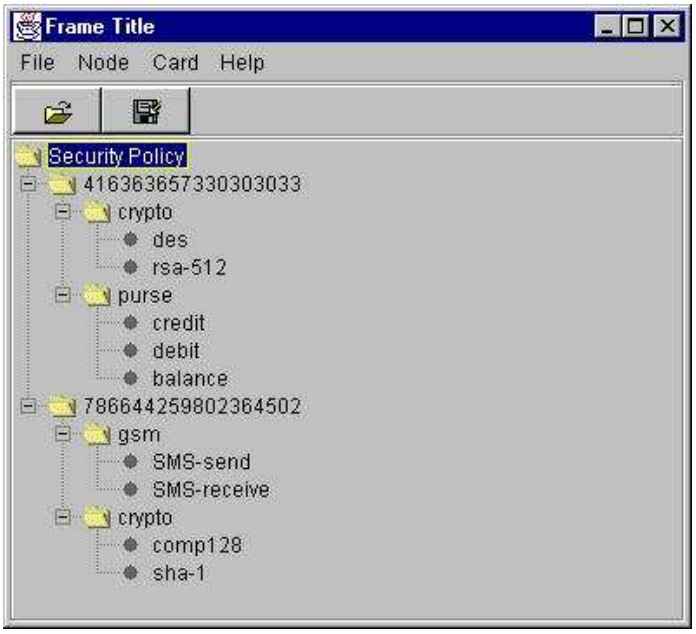

Fig. 3. The client for policy management

In the near future, we intend to improve it and to add other features such as stack introspection and a more complex code source including partial AID and digital signatures.

Acknowledgments

The authors are grateful to Jean-Luc Giraud for his corrections, suggestions and improvements.

References

1. Christophe Bidan, Pierre Bieber, and Pierre Girard. Trojan horses in multiapplicative smart cards. In *Submited to e-Smart 2001*, 2001.
2. Christophe Bidan and Pierre Girard. How to develop export-oriented smart solutions? In *Gemplus Devlopers Conference*, Paris, France, June 1999.
3. Visa. *Open Platform, Card Specification*, April 1999. Version 2.0.