A Simple Methodology for Secure Object Sharing

Daniel Perovich   Leonardo Rodríguez   Martín Varela
<perovich@fing.edu.uy> <lrodrigu@fing.edu.uy> <mvarela@fing.edu.uy>
Instituto de Computación
Facultad de Ingeniería
Universidad de la República
October 2000

1 Introduction

Smart cards have been in use since the early seventies, and their applications have evolved as a result of technological advances. JavaCards are a class of smart cards that have a special Java Virtual Machine (JCVM, for JavaCard Virtual Machine) embedded. The range of applications varies from healthcare or digital wallets to loyalty programs or access control. As the JavaCard technology spreads, new applicative areas for JavaCards (and smart cards in general) are considered.
JavaCards allow more than one application (called Applets) to coexist in them. This makes them very attractive, as the user can have, for example, a banking application, an e-wallet and his driving license in the same card.

When an applet is installed, it is given an AID (Applet Identifier as defined in ISO 7816-5)[ZHI], which is unique. When a company wants to deploy an applet, it must obtain an AID for it from the ISO. In the card, applets exist in Applet Contexts, which are isolated object spaces where all the objects of a certain package coexist. The *JavaCard Runtime Environment* (JCRE) [SUN1] enforces the object space isolation by means of the Applet Firewall. The firewall prevents objects in a certain context from directly accessing objects in another context. The JavaCard specification 2.1 provides applet developers with a way to share data and services between applets. This is called object sharing.

In this paper, we put forward a methodology for secure object sharing on the JavaCard platform. Our proposal is inspired by the work of Montgomery and Krishna, from Schlumberger [MONKRI]. That work is concerned with object sharing and proposes an approach to solve some of the problems that arise in the object sharing model proposed in the JavaCard 2.1 specification [SUN1]. Their work suggests some modifications to the JCRE specification as a possible solution to those problems. We base our approach on a methodology rather than on changes to the specification.

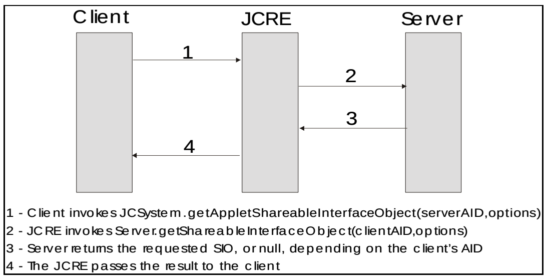
The next section comments on the object sharing model of JavaCard 2.1 specification and on Schlumberger's approach.
In section 3 we present the proposed methodology. The case study is described in section 4. Section 5 presents some comments on our experiences with a JavaCard emulator (Sun's JCWDE), and the conclusions are presented in section 6.

2 Object Sharing

In this section we present the JavaCard 2.1 specification object sharing model.
The mechanism proposed has several flaws, which we will present, along with the solution presented in [MONKRI].

2.1 The JavaCard 2.1 Object Sharing Model

In JavaCards, each applet exists within its own *context*, along with other objects from its package, and an applet cannot directly access objects in other applet's context. This makes for increased data security, but it limits the degree of interaction between applets. In earlier specifications of the JavaCard platform, the only way for applets to share data was by means of files. These files were protected by access control lists [MONKRI]. JavaCard 2.0 specification [SUN2] introduced the concept of object sharing, but with very important restrictions [SUN3]. The JavaCard 2.1 specification introduces the notion of *Shareable Interface*, which defines a set of methods that an applet may export through the firewall. If a developer wants certain methods in his applet (the server) to be exported, he must declare them in an interface that extends the tagging interface `javacard.framework.Shareable.` This tells the JCRE that these methods can be accessed from other contexts. Then, the developer must implement the defined interface (which is called a *Shareable Interface*) in a class, and instantiate an object of that class to obtain a *Shareable Interface Object* (SIO). When another applet (client applet) wants to access the exported methods, it must first declare a reference of the type defined by the server's Shareable Interface, and then invoke the `JCSystem.getAppletShareableInterfaceObject` method, indicating the AID (application identifier) of the server applet [SUN1, SUN4] The JCRE responds to this by invoking the server applet's `getShareableInterfaceObject`, (this method is defined in the `javacard.framework.Applet` abstract class, from which all applets descend), indicating the AID of the client applet. The server applet then decides, based on the client's AID, if the client is authorized to access the required SIO, and returns either a reference to that SIO, or null.



1 - Client invokes JCSystem.getAppletShareableInterfaceObject(serverAID,options)
2 - JCRE invokes Server.getShareableInterfaceObject(clientAID,options)
3 - Server returns the requested SIO, or null, depending on the client's AID
4 - The JCRE passes the result to the client

Figure 1. JavaCard 2.1 Object Sharing Mechanism.

This is just an outline of the object sharing mechanism, and the reader should refer to the JCRE specification [SUN1] for details.

2.2 Problems with Object Sharing

As described in [MONKRI], the object sharing model proposed for JavaCards has serious weaknesses and limitations.
If the client's authorization to obtain a SIO is based on its AID, a malicious applet could be installed on a compromised card, with the same AID as a valid client, and thus gain access to restricted data. Although there should be security policies to prevent this kind of attack [GIR], it might not be entirely impossible to carry out. Another problem with this selection criterion is that the server applet must know the AID of every possible client, which would make impossible to allow access to new clients once the server applet has been deployed.
It is also possible for a client applet to access a SIO for which it is not authorized. This happens only if an object implements more than one shareable interface, for example A and B. When this is the case, nothing prevents a client authorized to access shareable interface A from casting the reference it gets to shareable interface B, for which it may not be authorized.
Finally, this mechanism does not allow using objects passed as parameters in a shareable method. This is because when a method declared in a shareable interface is invoked, a context switch takes place, and the firewall prevents the server applet from accessing the object received as a parameter (see [SUN1] from more details on contexts and context switching).

2.3 Schlumberger's Delegate Object approach

In [MONKRI], a solution for some of the problems that arise in the current object sharing model is presented.
The proposed solution is based on the existence of *delegate objects*. In this approach, when an applet is registered, it can also register a delegate object, which will act as its interface with other applets. This object would manage all the interaction that the applet has with other applets. Access to delegate objects would be granted to any applet requesting it, and the delegate object manages all security issues. The security mechanism proposed is the use of a secret key and a challenge/response sequence, on a per method basis. This allows two of the problems that are present in the current model to be solved: the applet impersonation problem is solved (or at least partially solved) by the challenge/response mechanism. The limitation in the number of client AIDs accepted is also solved, since any client knowing the secret key, can pass the challenge posed by the server's delegate object, and gain access to the desired methods.
Furthermore, the secret key may be different for each method or group of related methods, and thus the client can be limited to a specific set of methods, based on the secret keys that it knows.

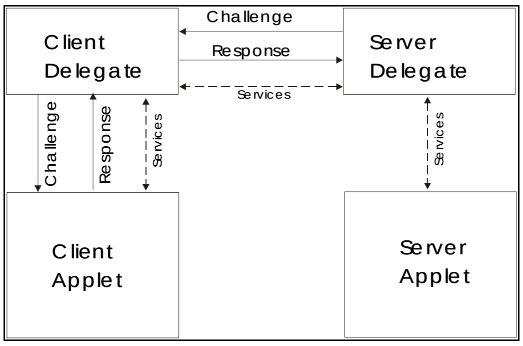


Figure 2. The delegate approach.

As all shareable methods are managed by the delegate object, a client applet cannot gain unauthorized access to any method by casting, as it happens in the current model.

The only issue that remains unsolved is the impossibility of passing objects as parameters.

3 A Methodology for Secure Object Sharing

The delegate object model proposed in [MONKRI] is not the only way to solve the problems that the JavaCard 2.1 object sharing model presents.

Modifying the JavaCard specification to conform to the delegate object model is a very significant change, and it could work badly with systems based on the current specification. Some correctness criteria should be specified and verified for the proposed model, so as to make sure that the changes introduced will not affect the rest of the JCRE.

However, some of the ideas behind the delegate object approach can be implemented under the current specification, based on a development methodology, which we now proceed to present.

3.1 Overview

The basic idea behind our approach is that, in order to obtain a SIO, a client must first register itself with the server. The registration process includes an authorization process, based on a challenge/response sequence. The registration lasts, at most, for the rest of the Card Acceptance Device (CAD) session. This is to prevent the substitution of a valid client applet for a malicious one, once the client has registered itself.

Once the client is registered, it can obtain the SIOs it needs.

This methodology allows for many different implementations, which may vary depending on the security needs of the application, the number of SIOs that the server offers and so on. We will discuss some examples of this later.

3.2 Basic Components

Our methodology is based on the existence of an object in the server package, which we call *SecureSIO*. This object is an instance of an implementation of a sharable interface, called *SecureSI*. This interface, in turn, provides methods that allow a client to prove its authenticity in order to obtain the required SIOs.

The basic implementation might be improved by using an *AuthorizationManager* (AMgr), which keeps record of all registered clients together with the SIOs they can access during a session. Both a SecureSIO and an AMgr manage all the security issues within the server. A new method is added to the SecureSI by means of which a client can unregister itself after it has finished using a SIO, so as to allow the AMgr, which has limited space for registration information, to accept more entries.

3.3 How does this work?

When a client wants to obtain a SIO from a server, it invokes the method `JCSystem.getAppletShareableInterfaceObject`, as it would normally do. As mentioned in section 3.1, this JCRE method in turn invokes the server's `getShareableInterfaceObject` method. This method should be redefined in the server to act as follows: the server queries the AMgr to verify that the client is authorized to obtain the SIO it is asking for. If the client is authorized, the server then returns the corresponding SIO. Otherwise, the server returns the SecureSIO, to allow the client to register itself. The client then proves its authenticity, and asks again for the desired SIO.

To obtain authorization, the client must ask the SecureSIO for a challenge, which will eventually depend on the SIO it is asking for. Once the client gets the challenge, it must provide a response for it, and if that response is correct, the SecureSIO proceeds to register it with the AuthorizationManager. It is important to note that in every moment, it is the client that initiates the communication with the server. This is to prevent a malicious applet from posing the client different challenges, and using it as a translator.

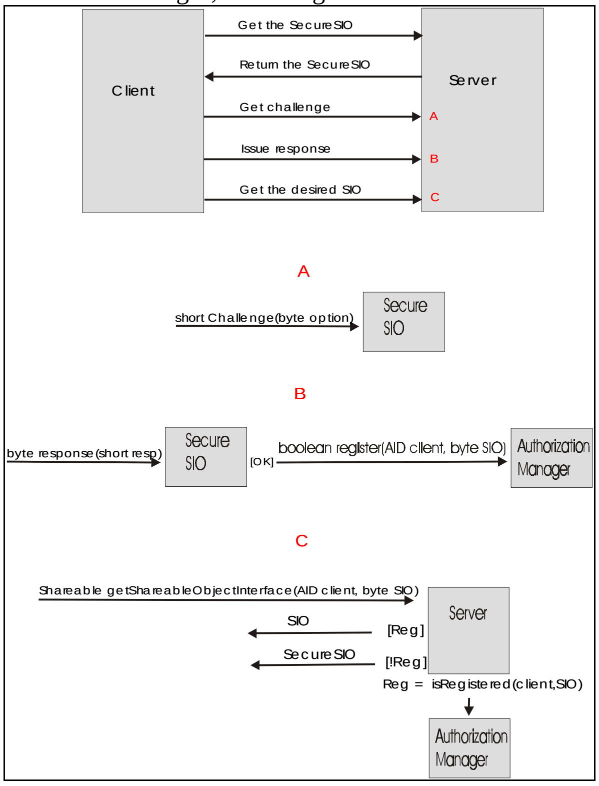


Figure 3. The proposed Object Sharing Mechanism.

Making the client start all communications does not make the hack mentioned above completely impossible, but it makes it harder to implement, since the client must be externally stimulated to make it ask for a certain SIO, and then use it to translate a secret key.
Given that the number of registered clients in the AuthorizationManager is limited, and in some cases, there could be many applets interacting with the server, the client may unregister itself to free space in the list of registered clients. In order to do this, it requires the SecureSIO to the server, and uses the `unregister` method it provides.

We now proceed to comment on some implementation issues that must be considered for the system to work as desired.

3.4 Applet Impersonation and Casting

Inappropriate casting is feasible only when several shareable interfaces are implemented in a single class. A way around this problem is to implement each shareable interface on a separate class, and to make sure these classes are not in the subclassing relation. This makes it impossible for the client to cast the SIO it obtains to another shareable interface [MONKRI].

As to impersonation, the AMgr is required to store the session's authorizations in CLEAR_ON_RESET transient objects, so that a client that has been registered during a session cannot be replaced with a malicious one for the next CAD session.

In addition, the AuthorizationManager can be implemented so that it stores authorizations for clients at method level, thus achieving the same granularity that delegate objects provide.

4 A Small Case Study

We now turn to present an experiment we developed using the *JavaCard Workstation Desktop Environment* (JCWDE), SUN's JavaCard 2.1 emulator.

We implemented a small and very simple server, which offers two different SIOs, and interacts with a number of clients. The server applet itself does nothing but receive requests from the clients. All the services offered are implemented in the SIOs, which reduces the server's logic to its `getShareableInterfaceObject` method.
The two SIOs offered are a small wallet, which exports three methods and a simplified medical record, which exports two methods.

The interaction between the server and the clients is as depicted in figure 3. When the client wants a certain SIO, it asks the server for it. The implementation of the server's

```
public Shareable
getShareableInterfaceObject(AID client,
byte sio){
        switch(sio){
              case SECURE_SIO:
                           return sSIO;

              case SHAREABLE_POCKET: {
                           if
(mgr.isRegistered(client,sio)){
                              return pocket;
                           }
                           else{
                              return sSIO;
                           }

              }
              case SHAREABLE_DATA: {

   if(mgr.isRegistered(client,sio)){
                              return data;
                           }
                           else{
                              return sSIO;
                           }

              }
              default: return null;
        }

}
```

Figure 4. Server's getShareableInterfaceObject method.

`getShareableInterfaceObject` method enforces the mechanism proposed, by checking with the AMgr (mgr) to verify whether the client is registered for that SIO or not.

If the client is registered, the method returns the appropriate SIO (be it data, or pocket). In case the client is not registered, it returns the SecureSIO (sSIO). In this implementation, the method returns `null` if an invalid SIO is asked for.

In this small example, the security issue is outlined, but not properly solved. The SecureSIO does not use any cryptographic protection, as it should. We left it out in order to keep the example simple.
When a client invokes the `register` method on the SecureSIO, it can obtain three possible results, depending on both the response provided and the space available in the Authorization Manager. Although the methodology allows for many clients requesting many SIOs from a single server in a single CAD session, most applications probably won't use that much applet interaction, and so the AMgr's space limitations shouldn't be an issue.

```
public class SecureSIO implements SecureSI {
   short currentChallenge = -1;
   byte currentSIO = 0;
   private AuthorizationManager mgr = null;


   public SecureSIO(AuthorizationManager amgr){
      mgr = amgr;
   }


   public short challenge(byte sio){
      currentSIO = sio;
      currentChallenge = sio;
      return currentChallenge;

   }

   public byte response(short resp){
      if(responseOk(resp)){

   if(mgr.register(JCSystem.getPreviousContextAID(),curre
ntSIO)){
            return SecureSI.RESPONSE_OK;
         }
         else{
            return SecureSI.NO_ROOM;
         }
      }
      else{
         return SecureSI.RESPONSE_FAILED;
      }

   }

   public void unregister(byte sio){
      mgr.unregister(JCSystem.getPreviousContextAID(),
sio);
   }


   private boolean responseOk(short resp){

      return (resp == currentChallenge);

   }
}
```

Figure 5. The SecureSIO class.

The implementation of the AuthorizationManager consists of a pair of arrays, containing a list of client AID's, and a list of SIOs, which are managed in parallel. Each pair of elements represents an entry to the AuthorizationManager.

```
public class AuthorizationManager {
  private Object[] currentClients = null;
  private byte[] authorizedSIOs = null;
  private byte currentAmount = 0;

protected AuthorizationManager(byte amount)
throws SystemException {
    byte i;
    currentClients =
JCSystem.makeTransientObjectArray(amount,
JCSystem.CLEAR_ON_RESET);
    authorizedSIOs =
JCSystem.makeTransientByteArray(amount,
JCSystem.CLEAR_ON_RESET);
    for(i = 0;i< currentClients.length;
i++){
      currentClients[i] = null;
    }
  }

public boolean isRegistered(AID aid, byte
sio){
    byte i = 0;
    while(i<currentAmount){
      if (aid.equals(currentClients[i]) &&
sio==authorizedSIOs[i]){
        return true;
      }
      else {
        i++;
      }
    }
    return false;
  }

public void unregister(AID aid, byte sio){
    byte i = 0;
    while(i<currentAmount){
      if (aid.equals(currentClients[i]) &&
authorizedSIOs[i++] == sio){
        authorizedSIOs[i] =
authorizedSIOs[currentAmount - 1];
        currentClients[i] = currentClients[-
-currentAmount];
        break;
      }
    }
}

public boolean register(AID aid, byte sio){
  if (isRegistered(aid,sio)) return true;
    if(currentAmount==currentClients.length)
    {return false;
    }
    else{
      authorizedSIOs[currentAmount] = sio;
      currentClients[currentAmount++] = aid;
      return true;
    }
  }

}
```

Figure 6. The AuthorizationManager class.

5 Notes on our experience with Sun's JCWDE

All the implementations that we have done so far have been developed with the Sun's JavaCard Development Kit 2.1. This kit provides tools for converting and verifying class files, emulating a JavaCard, and testing applets with APDU scripts. As to simulation itself, it provides the JCWDE, which is a (limited) card simulator, and the APDUtool, which is an APDU scripting tool.
Based on our experience with this kit, we believe that it needs some improvements. Firstly, and as described in the JavaCard Development Kit Release Notes [SUN5], there are core aspects of a JavaCard that have been left out, such as the firewall, and the impossibility of simulating a card reset.
Secondly, there is no easy way of creating APDU scripts, which must be written as strings of hex numbers. This could be easily improved, and it would save a great deal of time and errors.

6 Conclusions

We presented a methodology that may help developers to avoid some of the problems that arise when using the JavaCard 2.1 object sharing model.
The methodology proposed is simple to implement, and very flexible.
Although the overhead introduced by the use of the methodology may be greater than that of the delegate object approach, it does not require any changes in the JavaCard specification, which allows developers to use it with the existing implementations of JavaCard 2.1.
Since we only tested the methodology on a very simple case, and running on an emulator, there is still a lot of work to do, starting by implementing it on a real system, and on more complex cases.

7 References

- [MONKRI] Montgomery, M. and Krishna, K. - Secure Object Sharing in Java Card. Workshop on Smartcard Technology (Smartcard '99), USENIX, May 1999.

- [ZHI] Zhiqun Chen - How to write a Java Card Applet: A developer's guide http://www.javaworld.com/javaworld/jw-07-1999/jw-07-javacard.html

- [SUN1] Java Card 2.1 Runtime Environment (JCRE) Specification. Sun Microsystems, 1999. http://www.javasoft.com/products/javacard/

- [SUN2] JavaCard 2.0 Language Subset and Virtual Machine Specification. Sun Microsystems, 1997. http://www.javasoft.com/products/javacard/

- [SUN3] JavaCard 2.0 Programming Concepts. Sun Microsystems, 1997. http://www.javasoft.com/products/javacard/

- [SUN4] JavaCard 2.1 Application Programming Interface. Sun Microsystems, 1999. http://www.javasoft.com/products/javacard/

- [SUN5] JavaCard 2.1 Development Kit Release Notes. Sun Microsystems, 1999. http://www.javasoft.com/products/javacard/

- [GIR] Girard, P. - Which Security Policy for Multiapplication SmartCards Workshop on Smartcard Technology (Smartcard '99), USENIX, May 1999.

8 Acknowledgements

We would like to thank all the people who helped us in the process of writing this paper, especially Eduardo Giménez and Gustavo Betarte.