Secure Object Sharing in Java Card

Michael Montgomery
Austin Product Center
Schlumberger
Austin, TX 78726
`mmontgomery@slb.com`

Ksheerabdhi Krishna
Austin Product Center
Schlumberger
Austin, TX 78726
`kkrishna@slb.com`

Abstract

Since the invention of the Java Card, the issue of code and data sharing has been a topic of great interest. Early Java Cards shared data via files secured with access control lists. Java Card 2.1 specification introduced a method of object sharing, allowing access to methods of server applets using Shareable Interface Objects (SIO).

However, this SIO approach can be improved. It permits access to all interfaces of the SIO, whereas some interfaces may be intended only for particular clients. AID impersonation could be used to gain access to services unless the card authenticates all applets. Access to a SIO by future applets may be impossible. Passing object data between applets is quite cumbersome.

An approach to object sharing based on delegates is described, which provides needed improvements with minimal modifications to Java Card 2.1. Using the delegate approach, only the desired methods of an applet are exposed, and each method can be protected by any security policy the applet wishes to implement. A shared secret security policy is described, using challenge/response phrases to avoid revealing the shared secret. Such a security policy does not require applet authentication to avoid AID impersonation, and lends itself readily to access by any future applets that may be written.

1 Introduction

Since the invention of the Java Card, the issue of code and data sharing has been a topic of considerable interest. The first Java Cards [2] shared data between Java Card applets using a file system secured by access control lists. These lists determined which identities could access particular files, and what permission each identity was granted with respect to each file. The identities were established using key files and PIN verification. This solution was quite powerful for many common data sharing situations; however it did not lend itself well to situations requiring access to methods belonging to another Java Card applet.

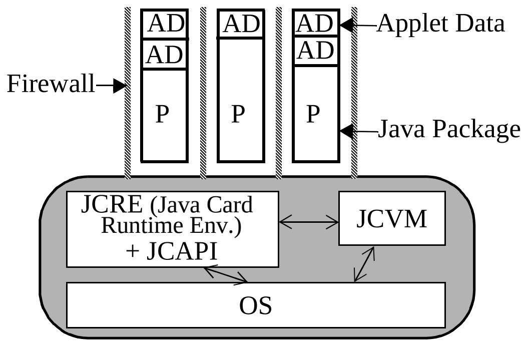


Figure 1: Basic Java Card Architecture

A typical Java Card architecture is shown in Figure 1. The card contains an operating system, Java Card Virtual Machine [7], Java Card Runtime Environment (JCRE), and the Java Card API in ROM. Applications are Java packages programmed to the API, and are usually loaded into EEPROM. An application consists of one or more applets; applets in different packages are separated by a firewall to prevent access to applet data across package boundaries. Applets are programmed using a subset of Java, and have a specified set of entry points that trigger various actions on the card [6].

In this paper, we will describe the object sharing mechanism introduced in Java Card 2.1, examine the issues associated with this mechanism, and propose alternatives that address these issues. We rely heavily on code examples, key elements of which are inserted into the paper at relevant points. Sometimes the code in the paper is edited for brevity, and comments containing "..." indicates the removal of code not pertinent to the discussion. The complete Java source code for these examples can be found in *http://www.cyberflex.slb.com/usenixMK99.html*. Note that when discussing the delegate examples, this code uses a framework which is suitable for simulation on a workstation, and is not intended for use on a Java card.

2 Object Sharing In Java Card 2.1

The Java Card 2.1 specification [10] introduces a means of sharing objects between Java Card applets. Although somewhat more difficult to use than file sharing, it does provide a means for accessing object methods, rather than just data. This specification uses a unique applet identifier (AID) [4] as the basis for determining which applets are granted access to objects created by other applets.

2.1 Description Of Object Sharing Mechanism

The strict firewall enforcement of Java Card 2.1 completely prevents an applet from accessing data corresponding to another applet. However, a provision was made for an applet to obtain an interface belonging to another applet, and to invoke a method on this interface. This forms the basis of the Java Card 2.1 object sharing mechanism.

2.1.1 Restricting Method Access through a SIO

Normally in Java [1], it would be possible to access public methods across packages. Therefore, one could use public methods in other packages without a need for a sharing mechanism.

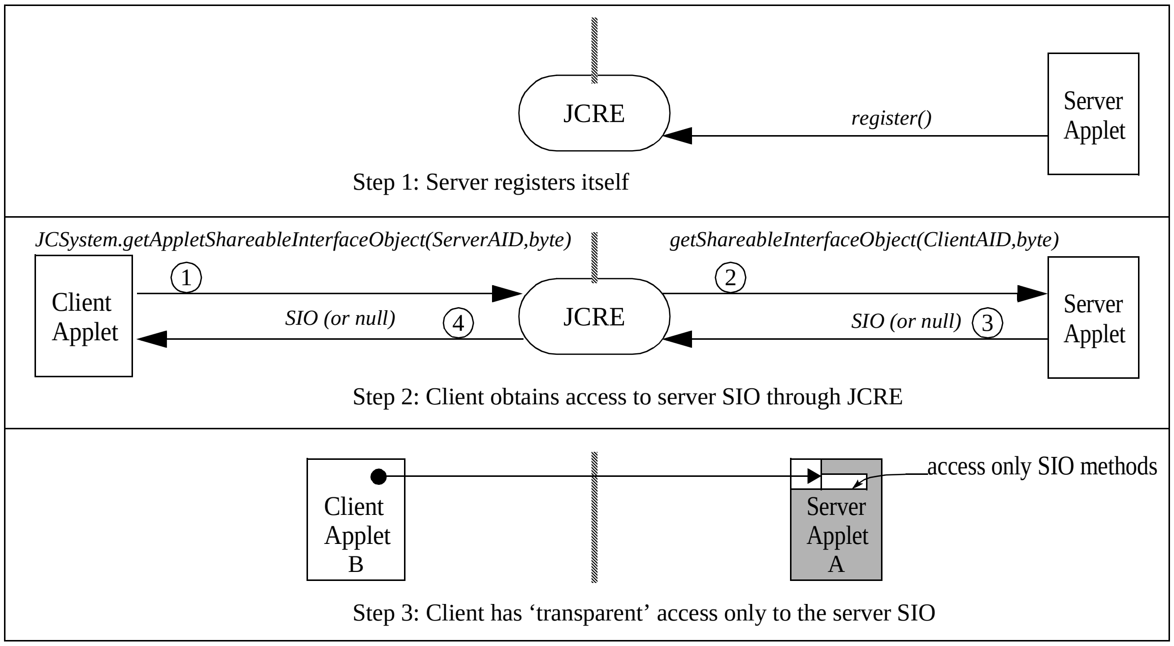
But this poses a problem for Java card, because the applet entry points are necessarily public, yet we must restrict access to them to prevent applets from running other applet methods without permission. So the JCRE does not permit any method to be invoked in other applets, except through the SIO mechanism. This prevents obvious hacks, such as casting an object reference to gain access to all of the public object methods, bypassing the restriction of the interface.

2.1.2 Applet Context

The object system in a Java Card is partitioned into separate protected object spaces referred to as contexts. All applets in a given package share the same context, and are prohibited from accessing objects in a different context due to firewalls which are enforced by the Java Card Runtime Environment (JCRE) [8]. The JCRE is able to access objects in any context, and global arrays such as the APDU buffer (which are owned by the JCRE) can be accessed by applets in any context.

2.1.3 Shareable Interface Objects (SIO)

Shareable interfaces are a new feature in the Java Card 2.1 API [9] to enable applets to explicitly share objects by defining a set of shared interface methods. Such shareable objects are called Shareable Interface Objects (SIO). We can think of those applets which provide SIO as server applets (since they will provide access to their services via the SIO), and those applets which use the SIO of another applet as client applets. Note that an applet may be a server to some applets, and yet a client of other applets.



JCRE
register()
Server Applet
Step 1: Server registers itself
JCSystem.getAppletShareableInterfaceObject(ServerAID,byte)
getShareableInterfaceObject(ClientAID,byte)
1
2
Client Applet
SIO (or null)
4
JCRE
SIO (or null)
3
Server Applet
Step 2: Client obtains access to server SIO through JCRE
access only SIO methods
Client Applet B
Server Applet A
Step 3: Client has ‘transparent’ access only to the server SIO

Figure 2: Creating and accessing a SIO

2.1.4 Creating a SIO

In order to create a new SIO, a server applet A must first define a shareable interface X, which extends the interface javacard.framework.Shareable. Applet A then defines a class C that implements the shareable interface X, and creates an instance O of class C. Object O is now a SIO. An instance of applet A with an AID is registered with the JCRE, as shown in Figure 2, step 1; this AID is subsequently used by client applications to specify the server applet.

2.1.5 Obtaining a SIO

A client applet must obtain this SIO in order to access object O. The process of a client obtaining an SIO is shown in Figure 2, step 2.

In order for client applet B to access object O, applet B must create an object reference BO of type X, and then call a system method getAppletShareableInterfaceObject with the AID of the server, and an optional byte which selects which interface is desired (for those servers with more than one interface available). The JCRE looks up the server applet associated with that AID, and forwards the request to the server, replacing the first argument with the client AID. Server applet A receives the request and the AID of the requester (applet B), and determines whether or not it will share object O with applet B. If applet A finds the request agreeable, then a reference to O is provided; otherwise, null is returned. The JCRE forwards this reference to Applet B. Applet B receives this reference (which is of type SIO), casts it to type X, and stores it in BO.

2.1.6 Using a SIO

Once a SIO has been obtained, applet B can invoke any methods from interface X on object reference BO, which then accesses object O. However, this is not as straightforward as it might seem, due to the firewalls.

Figure 2, step 3 shows the client applet B on the left, and the server applet A on the right, with a firewall in between. Note that the only part of applet A that is visible through the firewall is object O. When applet B invokes a method on BO, a context switch is triggered in the JCRE, leading to the situation in Figure 3. At this point, applet B is not visible at all; the only data visible from B are the arguments passed on the stack (and the global APDU array).

Note than it is pointless to pass object references on the stack, since the firewall will prevent object O (in Applet A) from using them, due to the context switch. However, object O can access any allowed data in Applet A as shown in Figure 3.

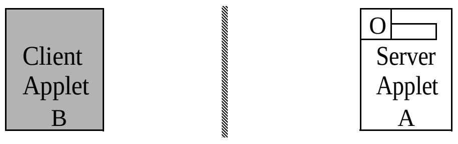


Figure 3: Server has no access to client data

2.2 Object Sharing Issues

There are four issues of concern with object sharing in Java Card 2.1.

2.2.1 Access To All Interface Methods Of A Class

The JCRE protection mechanisms do not prevent interfaces from being maliciously cast into other kinds of interfaces that might exist for a given object.

Once granted any interface, it is possible to access all of the shareable interface methods of an object via an explicit cast (not just the interface granted to a particular client). So if my server applet has two interfaces, intended for different kinds of clients, a client who legitimately obtains one interface, could cast it into the other interface, and gain access to unintended methods.

For example, suppose an applet ABCLoyaltyApplet has two different services that it intends to offer exclusively to two different clients, i.e. grantFequentFlyerPoints for client Airline, and grantLoyaltyPoints for client Purse. Both of these services are accessible to respective clients via different published interfaces, but both are defined in the Applet class, as illustrated in Code 1.

```
package AppABC;
import javacard.framework.*;

public interface ABCLoyaltyAirlineInterface extends Shareable {
   public void grantFrequentFlyerPoints(int amount);
}

public interface ABCLoyaltyPurseInterface extends Shareable {
   public void grantLoyaltyPoints(int amount);
}

public class ABCLoyaltyApplet extends
   javacard.framework.Applet implements
   ABCLoyaltyAirlineInterface, ABCLoyaltyPurseInterface {

   protected int loyaltyPoints;
   protected int frequentFlyerPoints;

   /* A service only for client 'Airline' */
   public void grantFrequentFlyerPoints(int amount) {
         frequentFlyerPoints += amount;
   }

   /* A service only for client 'Purse' */
   public void grantLoyaltyPoints(int amount) {
         loyaltyPoints += amount;
   }

   public ABCLoyaltyApplet() throws Exception {
      loyaltyPoints = 0;
      frequentFlyerPoints = 0;
      /* Add code to initialize potential client AIDs ... */
      register();
   }

   public void process() {
    /* Various code specific to services for this applet,
      such as point redemption code ... */
   }

   public Shareable getShareableInterfaceObject(
                  AID clientAID, byte parameter) {
      if (clientAID.equals(purseAppletAID))
         return (ABCLoyaltyPurseInterface) this;
      else if (clientAID.equals(airlineAppletAID))
         return (ABCLoyaltyAirlineInterface) this;
      else
         return null;
   }
}
```

Code 1: Server defines and grants access to SIO

If the Purse client was aware of the Airline client interface, it could make use of the interface it had been granted by the server applet to grant loyalty points, and instead use the grantFrequentFlyerPoints interface (intended for client Airline) to maliciously add unwarranted frequent flier points, as shown in Code 2.

```
package AppPurse;

import javacard.framework.*;
import AppABC.*;

public class PurseApplet extends javacard.framework.Applet {

   private int value;
   private ABCLoyaltyPurseInterface abcSIO;

   public PurseApplet() throws Exception {
      value = 0;
      abcSIO = (ABCLoyaltyPurseInterface)
         JCSystem.getAppletSharableInterfaceObject(
         abcLoyaltyAppletAID, (byte)0);
      register();
   }

   public void use(int amount) {
      value -= amount;
      if (abcSIO != null)
         abcSIO.grantLoyaltyPoints(amount);

      /* Attempt to 'hack' by using an SIO to access the Airline
       * applet's exclusive service grantFrequentFlyerPoints */
      ABCLoyaltyAirlineInterface hackSIO =
         (ABCLoyaltyAirlineInterface)
         JCSystem.getAppletSharableInterfaceObject(
         ABCLoyaltyAppletAID, (byte)0);
      if (hackSIO != null)
         hackSIO.grantFrequentFlyerPoints(amount);
   }
   /* Other methods follow for adding points to the purse ... */
}
```

Code 2: Client compromises SIO

Unfortunately, the JCRE is unable to prevent such access, since it does not violate the restriction of access only via shareable interfaces.

This problem can be eliminated by using separate delegate objects for each shared interface, and have the delegate objects handle or redirect the calls to the intended object. Another solution is to verify the AID of the caller upon each attempt to access a method in the server.

Furthermore, it is not possible to grant a client access to only certain methods of an interface. If such granularity is required, further delegates and interfaces must be used to separate the particular methods that are to be allowed for each applet. Alternatively, this granularity can be provided by having each method individually check the client AID, to determine whether the client should have access to that particular method.

2.2.2 AID Impersonation

The decision by a server applet to grant access to an object must be based solely on the AID of the requesting applet; no other information is available to the server applet. The intended use is clearly to allow certain interfaces to be granted only to particular client applets, as denoted by their unique AIDs. However, it is possible to maliciously set the AID of a rogue applet to be the same as the AID of a client applet known to have access to a particular interface. The rogue applet is then loaded *instead* of the applet that legitimately owns the AID. Once loaded, the rogue applet can request the desired interface. The server applet, having only the AID for reference, will naturally grant this request, since the AID matches the required AID for the interface. The rogue applet can then freely use the interface for malicious purposes.

This is a critical security problem. Current solutions to this problem require restrictions on applet loading, such as only allowing applets to be loaded that are signed by trusted sources. However, this approach greatly reduces the flexibility of Java Cards; for example, this prevents users from loading Java Card programs of their own devising.

2.2.3 Future Reference To Shared Objects

Granting access by AID necessarily assumes that the server applet that wishes to share the object has foreknowledge of all AIDs of all client applets that are to be loaded which will share particular interfaces. This is a difficult requirement for any kind of server. But what about other client applets that are written afterwards which legitimately need access to the shared object? Such applets are excluded from access to the object, since the server can only grant access based on the AID list that the server had when it was loaded. This necessitates rewriting and reissuing the server applet such that the new AIDs are included, which may pose an insurmountable hurdle when the server applet is already widely distributed.

2.2.4 Inability To Pass Object Parameters

The inability to pass object parameters between applets can make many tasks cumbersome. For example, supposes as a client needs to pass an object when invoking a method on a server applet so that the server can manipulate and return the object, as in encrypting a buffer. In this case, the object might contain a field specifying the method of encryption, a key array, and a data array. But as stated earlier, the firewall prevents the server applet from accessing this data, even if an object reference from the client is provided.

One might envision a work around to this problem where the server A gets a shared interface on an object in client B in order to read the object data as shown in Figure 4.

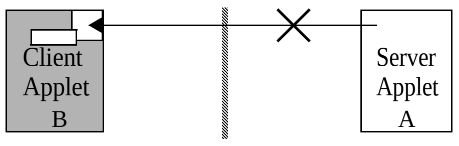


Figure 4: Server access blocked by firewall

However, this will fail, since any attempt by server A to get the object data from client B will also be blocked by the firewall. The server can only invoke methods on the client object, but the client object has no means for returning the object data, other than a single field at a time. Thus the only current work around to this problem is to use the global APDU buffer to pass data as shown in Figure 5.

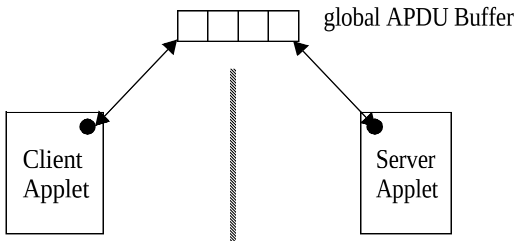


Figure 5: Data exchange via global APDU buffer

This is awkward at best, since the data being passed may not be of appropriate length or type to be readily passed via this mechanism. Consequently, the client and server might have to make multiple calls and considerable manipulations to pass the desired parameters, due to the restrictions of the APDU buffer.

This problem could be addressed by changing the firewall restrictions to permit access to shared objects (including data), instead of just interfaces; however, this could potentially create security holes. Applets must coded carefully to avoid exposing methods and data that are not intended to be shared. But if instead of actually sharing the applet object, the applet used a delegate to expose only the desired methods and data, the JCRE specification could be altered to permit access to only delegate methods and data, thus eliminating the object parameter problem, with a minimum of security risk.

3 Delegate Approach To Object Sharing

Based on an analysis of these critical problems, an approach to object sharing was devised which avoids these drawbacks. In this approach, each server applet that wishes to permit access to its methods or data creates a single delegate, which is registered with the system based on the AID of the applet. The delegate exposes only those methods and data that the applet wishes to share. Access to the delegate is public; all methods in the delegate can be accessed by any other applet!

Since access to the delegate is unrestricted, an applet protects itself in two ways. First, the delegate is written to only access the desired methods of the server applet; other applet methods cannot be accessed. Second, the methods of the delegate can perform any checks as deemed necessary to check the validity of the access to the delegate; if the checks are not passed, the delegate can refuse to pass the request on to the applet.

3.1 Java Card 2.1 System Changes Required

This approach presumes the addition of two new Java Card 2.1 system methods. A version of register is needed which takes a delegate and AID as arguments. A system method getDelegate returns the delegate associated with a particular AID. These methods replace the JCSystem.getAppletShareableInterfaceObject and Applet.getShareableInterfaceObject methods. Furthermore, the JCRE is altered to permit access to methods and data of delegate objects (DOs) in other contexts (just as it now permits access to methods of SIOs in different contexts).

For performance reasons, it would be worthwhile to investigate whether it is necessary for the JCRE to restrict access to objects in other contexts at all, since checking object context imposes a speed penalty. Techniques such as the delegate method proposed, coupled with classical Java language and runtime protections[11,5], could perhaps result in a better performing system that still meets the security requirements.

3.2 Delegate Server Implementation

A server applet must be written containing the desired methods. For this example, we show a loyalty server applet in Code 3 from the ABC company that allows granting, using, and reading loyalty points.

```
package AppABC;
import OSResources.*;

public class ABCLoyaltyApplet extends OSResources.Applet {
   private String aid = "AppABC.ABCLoyaltyApplet";
   private int loyalty;

   public ABCLoyaltyApplet() {
      loyalty = 0;
      register(new ABCLoyaltyDelegate(this), aid);
   }

   void grantPoints(int amount) {
         loyalty += amount;
   }

   boolean usePoints(int amount) {
      if (loyalty >= amount) {
         loyalty -= amount;
         return true;
      } else {
         return false;
      }
   }

   int readPoints() {
      return loyalty;
   }
}
```

Code 3: Server applet registering its delegate

Note that the only novel aspect of this applet is when it creates and registers its delegate, as illustrated in Figure 6, step 1. Whenever a server applet wishes to allow access to another applet, a delegate must be written which exposes the desired methods. Such a delegate is shown in Code 4.

```
package AppABC;
import OSResources.*;

public class ABCLoyaltyDelegate extends Delegate {

  final static byte CHALLENGE_LENGTH = (byte) 64;
  final static byte FAILURES_ALLOWED = (byte) 2;

  private byte[] secret1 = { /* initializing code ...*/ };
  private byte[] secret2 = { /* initializing code ...*/ };

  private ABCLoyaltyApplet myApplet;
  private ChallengePhrase cp;
  private ChallengePhrase checkcp;
  private byte grantTriesRemaining = FAILURES_ALLOWED;
  private byte useTriesRemaining = FAILURES_ALLOWED;

  protected ABCLoyaltyDelegate(ABCLoyaltyApplet a) {
     myApplet = a;
     cp = new ChallengePhrase(CHALLENGE_LENGTH);
     checkcp = new ChallengePhrase(CHALLENGE_LENGTH);
  }

  public boolean grantPoints(int amount) {
     // do some checking
     if (myApplet != null) {
        ABCLoyaltyInterface pD = (ABCLoyaltyInterface)
           OSystem.getDelegate(
              OSystem.getPreviousContextAID());
        if (pD != null) {
           /* Create new random challenge phrase */
           checkcp.setPhrase(cp.randomize());
           /* Get client response to phrase */
           byte[] response = pD.loyaltyChallenge(cp);
           byte[] r = checkcp.encrypt(secret1);
           if (isEqual(response,r)) {
              /* See if client gave the correct response */
              grantTriesRemaining = FAILURES_ALLOWED;
              myApplet.grantPoints(amount);
              return true;
           } else {
              if(--grantTriesRemaining == 0) myApplet = null;
              return false;
           }
        }
     }
     return false;
  }

  public boolean usePoints(int amount) {
  /* Access to this method uses secret2, for different
     clients, but is otherwise similiar to grantPoints ... */
  }

  public int readPoints() {
     if (myApplet != null)
        return myApplet.readPoints();
     else
        return 0;
  }
}
```

Code 4: Server delegate

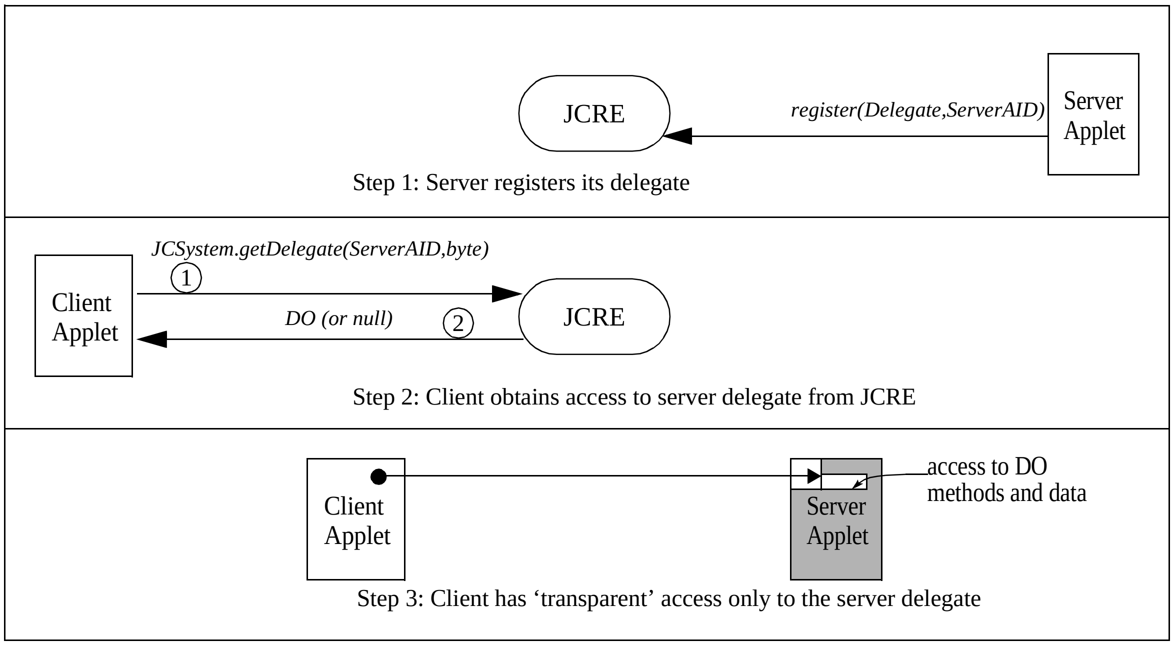

JCRE
register(Delegate,ServerAID)
Server Applet
Step 1: Server registers its delegate
JCSystem.getDelegate(ServerAID,byte)
1
Client Applet
DO (or null)
2
JCRE
Step 2: Client obtains access to server delegate from JCRE
Client Applet
access to DO methods and data
Server Applet
Step 3: Client has ‘transparent’ access only to the server delegate

Figure 6: Creating and accessing a delegate

This delegate is registered with the AID of the server applet at the time the applet is instantiated on the Java Card. At this point, this delegate is publicly accessible by any other applet. This particular delegate defines the interface ABCLoyaltyInterface shown in Code 5, which must be implemented by client applications. The reasons for this interface are discussed in detail in Section 3.3.

```
package AppABC;
import OSResources.*;

public interface ABCLoyaltyInterface{
   public byte[] loyaltyChallenge(ChallengePhrase cp);
}
```

Code 5: Server defines interface that each client must provide to handle challenge/response

3.3 Delegate Security

A delegate controls which methods each client can access. This delegate exposes three methods: grantPoints, usePoints, and readPoints. The method readPoints has no validation, and may be freely used by any client.

But what about the methods that require proof that the access is valid, such as grantPoints and usePoints? These methods could just check the AID of the client. This effectively mimics the object sharing of Java Card 2.1, which is subject to the risks of AID impersonation and difficulty of adding future clients.

These problems can be avoided by using the classical solution of shared secrets. In this example, access to grantPoints is controlled by secret1; only clients that can authenticate knowledge of secret1 will have their request passed from the delegate to the server applet. Access to usePoints is controlled by secret2, which allows a potentially different set of clients to access this method.

There are many standard techniques for handling shared secrets and other methods of authentication [3]. We are not proposing anything new here, other than the application of these techniques to this problem, to avoid the limitations and pitfalls of security based on AID. The details of the application of the shared secret technique to Java Card is presented in detail for the benefit of those who may not be familiar with such techniques, or may be unclear how they can be implemented in a Java card.

Proving knowledge of the secret could be a tricky proposition. Passing the secret as an argument to the server applet is a bad idea, since the secret could be intercepted in a number of ways. (One simple way to get the secret is to write a applet that impersonates the server applet, thus capturing the secret when it is passed as an argument.) A superior approach is to use challenge/response phrases, as illustrated by the server delegate in Code 4. When access is requested, the server delegate sends a random challenge phrase to a predefined method in the client delegate. This challenge phrase is encrypted using the secret by the client delegate, and returned to the server delegate. The server delegate performs the same encryption, and if the results match, access is granted. Thus the secret is never revealed outside of either applet.

3.4 Delegate Client Implementation

When a client applet wishes to use the server applet, it calls getDelegate with the AID of the server applet, and receives a delegate, which it casts to the proper delegate class associated with the server applet, as shown in Figure 6, step 2**.** The client applet then invokes methods on the delegate as desired. Note that unlike SIO, the JCRE hands out the delegate, and server applet is not involved. The client in Code 6 was written assuming authentication using challenge/response.

```
package AppPurse;

import OSResources.*;
import AppABC.*;

public class PurseApplet extends OSResources.Applet {

   public String aid = "AppPurse.PurseApplet";
   private byte[] secret1 = { /* initializing code ...*/ };
   private int value;

   public PurseApplet() {
      value = 0;
      register(new PurseDelegate(this), aid);
   }

   public byte[] loyaltyChallenge(ChallengePhrase cp) {
      return cp.encrypt(secret1);
   }

   public void use(int amount) {
      value -= amount;

      ABCLoyaltyDelegate d =
         (ABCLoyaltyDelegate)
         OSystem.getDelegate("AppABC.ABCLoyaltyApplet");
      if (d != null) {
         d.grantPoints(amount);
      }
   }
/* Other methods follow for adding points to the purse ... */
}
```

Code 6: Client applet invoking server method

To handle the challenge/response, the client must supply a delegate as shown in Code 7 that the server will call to perform the necessary encryption with the challenge phrase. This client delegate must implement the ABCLoyaltyInterface, as defined by the server applet. The delegate may not actually perform the encryption, but may instead pass the challenge/response request to the client applet for processing. This allows the delegate to avoid holding the shared secret, reducing the security risk.

```
package AppPurse;

import OSResources.*;
import AppABC.*;

public class PurseDelegate extends Delegate implements
ABCLoyaltyInterface{

   private PurseApplet myApplet;

   protected PurseDelegate(PurseApplet a) {
      myApplet = a;
   }

   public byte[] loyaltyChallenge(ChallengePhrase cp) {
      if (myApplet != null)
         return(myApplet.loyaltyChallenge(cp));
      else
         return null;
   }
}
```

Code 7: Client challenge/response delegate

3.5 Overview Of Client/Server Communication

At this point, it is presumed that the server has already registered with the JCRE, and that the client has already obtained the server delegate, as shown in Figure 6, and the client is ready to use a service from the server. The resulting client/server communication using a shared secret is illustrated in Figure 7.

The client begins by requesting a service from the server. This is done by invoking a method on the server delegate object ①. The method in the server delegate determines what level of protection is required; in this case, it determines that a shared secret must be validated to use this method. It therefore requests the delegate for the client from the JCRE ②, which the JCRE provides (if it exists) ③. Assuming the client delegate was successfully obtained, the server obtains a random challenge phrase, and sends it to the client delegate ④ using the interface the server had predefined for this purpose. The client delegate passes the challenge to the client applet (which contains the secret) ⑤. The client applet encrypts the challenge phrase using the shared secret, and returns the response phrase to the client delegate ⑥. The client delegate then passes the response phrase to the server delegate ⑦. The server delegate also encrypts the challenge phrase with the shared secret, and if the results match, the secret is validated. The server delegate then forwards the service request to the server applet ⑧, which processes the request, and returns a response to the delegate ⑨, which is forwarded to the client applet ⑩.

If the nature of the server applet is such that the services are likely to be reused by a given client in a single session, then after validation of the shared secret, the server would likely be designed to return a session key (which permits access only during the current session, which ends when power is cycled). The client could then access that particular service through the delegate by providing the session key, thus avoiding the overhead of validating the shared secret for each access of a delegate method. Although the session key is provided by the client to the server in clear text form, there is no real security risk, since the server randomizes the currently active session key each time power is cycled. So even if the session key were intercepted by an applet impersonating the server, it could not be used to breach security, due to the randomizing of the session keys.

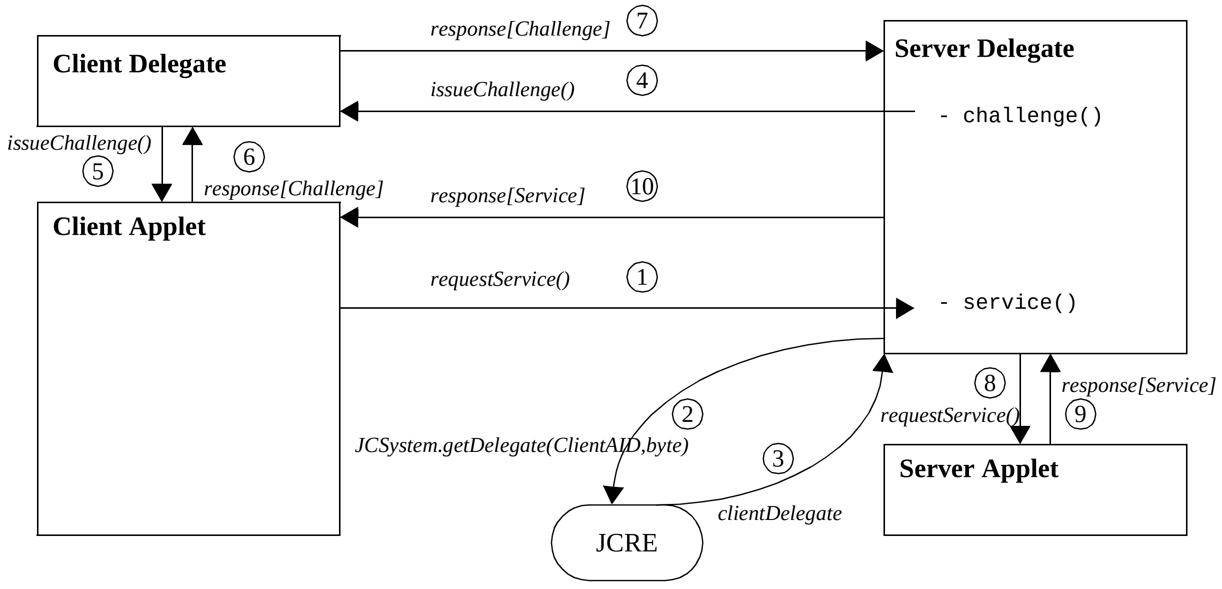


Figure 7: Challenge/Response validation of shared secret

4 Conclusion

The proposed delegate method of object sharing improves on the SIO approach. It protects each method as desired, so that no client applet gains access to unintended methods of a server applet. It avoids AID impersonation, since AIDs can be supplemented with shared secrets or other authentication mechanisms. It avoids future reference problems, since access need not be linked to any particular AID, but can simply be based on a shared secret which can easily be added to future client applications. Moreover, it requires only a minimal addition to the Java Card system. Since the delegate method allows each applet to determine the security policy desired on a per method basis, this allows the maximum flexibility for granularity of access by each individual client.

5 References

[1] Arnold, K. and Gosling, J., *The Java Programming Language*, Addison-Wesley, 1996.

[2] Guthery, Scott. B., *Java Card: Internet Computing On A Smart Card*, IEEE Internet Computing, pp. 57-59, Jan/Feb 1997.

[3] Guthery, Scott B., and Jurgensen, Timothy M., *Smart Card Developer's Kit*, Macmillian Technical Publishing, 1998.

[4] ISIO-7816, *Information Technology - Identification cards - integrated circuit cards with contacts.*

[5] McGraw, Gary E. and Felten, Edward W., *Java Security: Hostile Applets, Holes, and Antidotes*, John Wiley and Sons, 1996.

[6] McManis, Chuck. *My ENIGMatic Java Ring*. JavaWorld, 3(8), August 1998. http://www.javaworld.com/javaworld/jw-08-1998/jw-08-in-depth.html

[7] Sun Microsystems Inc., *Java Card 2.1 Virtual Machine Specification*, //java.sun.com/products/javacard/JCVMSpec.pdf

[8] Sun Microsystems Inc., *Java Card 2.1 Runtime Environment Specification*, //java.sun.com/products/javacard/JCRESpec.pdf

[9] Sun Microsystems Inc., *Java Card 2.1 Application Programming Interfaces Specification*, //java.sun.com/products/javacard/htmldoc/index.html

[10] Sun Microsystems Inc., *The Java Card 2.1 Platform Specifications*, //java.sun.com/products/javacard/

[11] Yellin, Frank., *Low level security in Java*, Fourth International World Wide Web Conference, Boston, MA, December 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html

Secure Object Sharing in Java Card

Michael Montgomery
Austin Product Center
Schlumberger
Austin, TX 78726
`mmontgomery@slb.com`

Ksheerabdhi Krishna
Austin Product Center
Schlumberger
Austin, TX 78726
`kkrishna@slb.com`

Abstract

Since the invention of the Java Card, the issue of code and data sharing has been a topic of great interest. Early Java Cards shared data via files secured with access control lists. Java Card 2.1 specification introduced a method of object sharing, allowing access to methods of server applets using Shareable Interface Objects (SIO).

However, this SIO approach can be improved. It permits access to all interfaces of the SIO, whereas some interfaces may be intended only for particular clients. AID impersonation could be used to gain access to services unless the card authenticates all applets. Access to a SIO by future applets may be impossible. Passing object data between applets is quite cumbersome.

An approach to object sharing based on delegates is described, which provides needed improvements with minimal modifications to Java Card 2.1. Using the delegate approach, only the desired methods of an applet are exposed, and each method can be protected by any security policy the applet wishes to implement. A shared secret security policy is described, using challenge/response phrases to avoid revealing the shared secret. Such a security policy does not require applet authentication to avoid AID impersonation, and lends itself readily to access by any future applets that may be written.

1 Introduction

Since the invention of the Java Card, the issue of code and data sharing has been a topic of considerable interest. The first Java Cards [2] shared data between Java Card applets using a file system secured by access control lists. These lists determined which identities could access particular files, and what permission each identity was granted with respect to each file. The identities were established using key files and PIN verification. This solution was quite powerful for many common data sharing situations; however it did not lend itself well to situations requiring access to methods belonging to another Java Card applet.

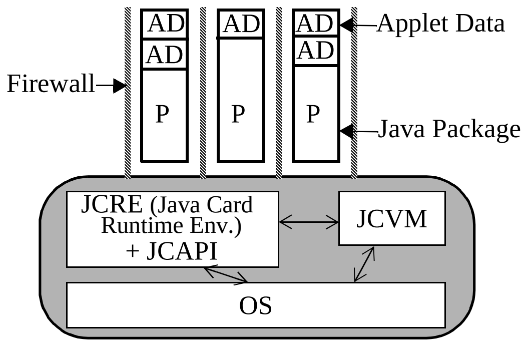


Figure 1: Basic Java Card Architecture

A typical Java Card architecture is shown in Figure 1. The card contains an operating system, Java Card Virtual Machine [7], Java Card Runtime Environment (JCRE), and the Java Card API in ROM. Applications are Java packages programmed to the API, and are usually loaded into EEPROM. An application consists of one or more applets; applets in different packages are separated by a firewall to prevent access to applet data across package boundaries. Applets are programmed using a subset of Java, and have a specified set of entry points that trigger various actions on the card [6].

In this paper, we will describe the object sharing mechanism introduced in Java Card 2.1, examine the issues associated with this mechanism, and propose alternatives that address these issues. We rely heavily on code examples, key elements of which are inserted into the paper at relevant points. Sometimes the code in the paper is edited for brevity, and comments containing "..." indicates the removal of code not pertinent to the discussion. The complete Java source code for these examples can be found in *http://www.cyberflex.slb.com/usenixMK99.html*. Note that when discussing the delegate examples, this code uses a framework which is suitable for simulation on a workstation, and is not intended for use on a Java card.

2 Object Sharing In Java Card 2.1

The Java Card 2.1 specification [10] introduces a means of sharing objects between Java Card applets. Although somewhat more difficult to use than file sharing, it does provide a means for accessing object methods, rather than just data. This specification uses a unique applet identifier (AID) [4] as the basis for determining which applets are granted access to objects created by other applets.

2.1 Description Of Object Sharing Mechanism

The strict firewall enforcement of Java Card 2.1 completely prevents an applet from accessing data corresponding to another applet. However, a provision was made for an applet to obtain an interface belonging to another applet, and to invoke a method on this interface. This forms the basis of the Java Card 2.1 object sharing mechanism.

2.1.1 Restricting Method Access through a SIO

Normally in Java [1], it would be possible to access public methods across packages. Therefore, one could use public methods in other packages without a need for a sharing mechanism.

But this poses a problem for Java card, because the applet entry points are necessarily public, yet we must restrict access to them to prevent applets from running other applet methods without permission. So the JCRE does not permit any method to be invoked in other applets, except through the SIO mechanism. This prevents obvious hacks, such as casting an object reference to gain access to all of the public object methods, bypassing the restriction of the interface.

2.1.2 Applet Context

The object system in a Java Card is partitioned into separate protected object spaces referred to as contexts. All applets in a given package share the same context, and are prohibited from accessing objects in a different context due to firewalls which are enforced by the Java Card Runtime Environment (JCRE) [8]. The JCRE is able to access objects in any context, and global arrays such as the APDU buffer (which are owned by the JCRE) can be accessed by applets in any context.

2.1.3 Shareable Interface Objects (SIO)

Shareable interfaces are a new feature in the Java Card 2.1 API [9] to enable applets to explicitly share objects by defining a set of shared interface methods. Such shareable objects are called Shareable Interface Objects (SIO). We can think of those applets which provide SIO as server applets (since they will provide access to their services via the SIO), and those applets which use the SIO of another applet as client applets. Note that an applet may be a server to some applets, and yet a client of other applets.

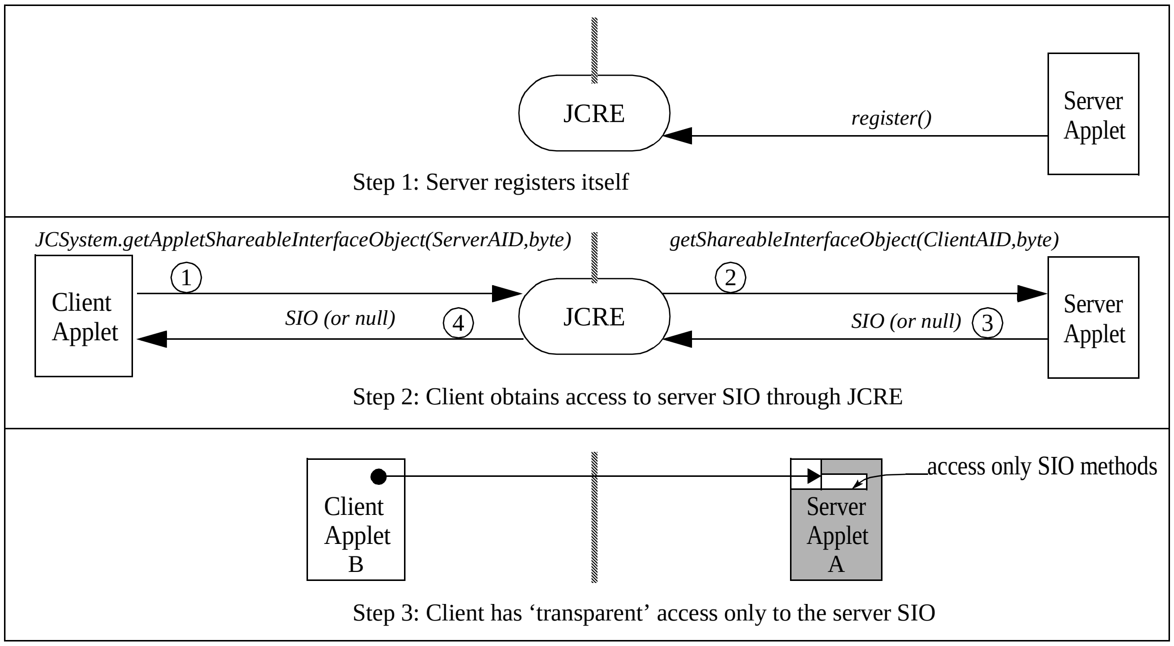

JCRE
register()
Server Applet
Step 1: Server registers itself
JCSystem.getAppletShareableInterfaceObject(ServerAID,byte)
getShareableInterfaceObject(ClientAID,byte)
Client Applet
1
2
JCRE
Server Applet
SIO (or null)
4
SIO (or null)
3
Step 2: Client obtains access to server SIO through JCRE
access only SIO methods
Client Applet B
Server Applet A
Step 3: Client has ‘transparent’ access only to the server SIO

Figure 2: Creating and accessing a SIO

2.1.4 Creating a SIO

In order to create a new SIO, a server applet A must first define a shareable interface X, which extends the interface javacard.framework.Shareable. Applet A then defines a class C that implements the shareable interface X, and creates an instance O of class C. Object O is now a SIO. An instance of applet A with an AID is registered with the JCRE, as shown in Figure 2, step 1; this AID is subsequently used by client applications to specify the server applet.

2.1.5 Obtaining a SIO

A client applet must obtain this SIO in order to access object O. The process of a client obtaining an SIO is shown in Figure 2, step 2.

In order for client applet B to access object O, applet B must create an object reference BO of type X, and then call a system method getAppletShareableInterfaceObject with the AID of the server, and an optional byte which selects which interface is desired (for those servers with more than one interface available). The JCRE looks up the server applet associated with that AID, and forwards the request to the server, replacing the first argument with the client AID. Server applet A receives the request and the AID of the requester (applet B), and determines whether or not it will share object O with applet B. If applet A finds the request agreeable, then a reference to O is provided; otherwise, null is returned. The JCRE forwards this reference to Applet B. Applet B receives this reference (which is of type SIO), casts it to type X, and stores it in BO.

2.1.6 Using a SIO

Once a SIO has been obtained, applet B can invoke any methods from interface X on object reference BO, which then accesses object O. However, this is not as straightforward as it might seem, due to the firewalls.

Figure 2, step 3 shows the client applet B on the left, and the server applet A on the right, with a firewall in between. Note that the only part of applet A that is visible through the firewall is object O. When applet B invokes a method on BO, a context switch is triggered in the JCRE, leading to the situation in Figure 3. At this point, applet B is not visible at all; the only data visible from B are the arguments passed on the stack (and the global APDU array).

Note than it is pointless to pass object references on the stack, since the firewall will prevent object O (in Applet A) from using them, due to the context switch. However, object O can access any allowed data in Applet A as shown in Figure 3.

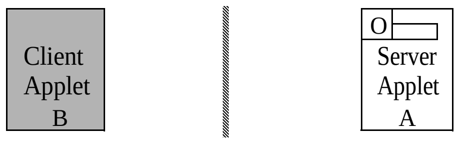


Figure 3: Server has no access to client data

2.2 Object Sharing Issues

There are four issues of concern with object sharing in Java Card 2.1.

2.2.1 Access To All Interface Methods Of A Class

The JCRE protection mechanisms do not prevent interfaces from being maliciously cast into other kinds of interfaces that might exist for a given object.

Once granted any interface, it is possible to access all of the shareable interface methods of an object via an explicit cast (not just the interface granted to a particular client). So if my server applet has two interfaces, intended for different kinds of clients, a client who legitimately obtains one interface, could cast it into the other interface, and gain access to unintended methods.

For example, suppose an applet ABCLoyaltyApplet has two different services that it intends to offer exclusively to two different clients, i.e. grantFequentFlyerPoints for client Airline, and grantLoyaltyPoints for client Purse. Both of these services are accessible to respective clients via different published interfaces, but both are defined in the Applet class, as illustrated in Code 1.

```
package AppABC;
import javacard.framework.*;

public interface ABCLoyaltyAirlineInterface extends Shareable {
   public void grantFrequentFlyerPoints(int amount);
}

public interface ABCLoyaltyPurseInterface extends Shareable {
   public void grantLoyaltyPoints(int amount);
}

public class ABCLoyaltyApplet extends
   javacard.framework.Applet implements
   ABCLoyaltyAirlineInterface, ABCLoyaltyPurseInterface {

   protected int loyaltyPoints;
   protected int frequentFlyerPoints;

   /* A service only for client 'Airline' */
   public void grantFrequentFlyerPoints(int amount) {
         frequentFlyerPoints += amount;
   }

   /* A service only for client 'Purse' */
   public void grantLoyaltyPoints(int amount) {
         loyaltyPoints += amount;
   }

   public ABCLoyaltyApplet() throws Exception {
      loyaltyPoints = 0;
      frequentFlyerPoints = 0;
      /* Add code to initialize potential client AIDs ... */
      register();
   }

   public void process() {
    /* Various code specific to services for this applet,
      such as point redemption code ... */
   }

   public Shareable getShareableInterfaceObject(
                  AID clientAID, byte parameter) {
      if (clientAID.equals(purseAppletAID))
         return (ABCLoyaltyPurseInterface) this;
      else if (clientAID.equals(airlineAppletAID))
         return (ABCLoyaltyAirlineInterface) this;
      else
         return null;
   }
}
```

Code 1: Server defines and grants access to SIO

If the Purse client was aware of the Airline client interface, it could make use of the interface it had been granted by the server applet to grant loyalty points, and instead use the grantFrequentFlyerPoints interface (intended for client Airline) to maliciously add unwarranted frequent flier points, as shown in Code 2.

```
package AppPurse;

import javacard.framework.*;
import AppABC.*;

public class PurseApplet extends javacard.framework.Applet {

   private int value;
   private ABCLoyaltyPurseInterface abcSIO;

   public PurseApplet() throws Exception {
      value = 0;
      abcSIO = (ABCLoyaltyPurseInterface)
         JCSystem.getAppletSharableInterfaceObject(
         abcLoyaltyAppletAID, (byte)0);
      register();
   }

   public void use(int amount) {
      value -= amount;
      if (abcSIO != null)
         abcSIO.grantLoyaltyPoints(amount);

      /* Attempt to 'hack' by using an SIO to access the Airline
       * applet's exclusive service grantFrequentFlyerPoints */
      ABCLoyaltyAirlineInterface hackSIO =
         (ABCLoyaltyAirlineInterface)
         JCSystem.getAppletSharableInterfaceObject(
         ABCLoyaltyAppletAID, (byte)0);
      if (hackSIO != null)
         hackSIO.grantFrequentFlyerPoints(amount);
   }
   /* Other methods follow for adding points to the purse ... */
}
```

Code 2: Client compromises SIO

Unfortunately, the JCRE is unable to prevent such access, since it does not violate the restriction of access only via shareable interfaces.

This problem can be eliminated by using separate delegate objects for each shared interface, and have the delegate objects handle or redirect the calls to the intended object. Another solution is to verify the AID of the caller upon each attempt to access a method in the server.

Furthermore, it is not possible to grant a client access to only certain methods of an interface. If such granularity is required, further delegates and interfaces must be used to separate the particular methods that are to be allowed for each applet. Alternatively, this granularity can be provided by having each method individually check the client AID, to determine whether the client should have access to that particular method.

2.2.2 AID Impersonation

The decision by a server applet to grant access to an object must be based solely on the AID of the requesting applet; no other information is available to the server applet. The intended use is clearly to allow certain interfaces to be granted only to particular client applets, as denoted by their unique AIDs. However, it is possible to maliciously set the AID of a rogue applet to be the same as the AID of a client applet known to have access to a particular interface. The rogue applet is then loaded *instead* of the applet that legitimately owns the AID. Once loaded, the rogue applet can request the desired interface. The server applet, having only the AID for reference, will naturally grant this request, since the AID matches the required AID for the interface. The rogue applet can then freely use the interface for malicious purposes.

This is a critical security problem. Current solutions to this problem require restrictions on applet loading, such as only allowing applets to be loaded that are signed by trusted sources. However, this approach greatly reduces the flexibility of Java Cards; for example, this prevents users from loading Java Card programs of their own devising.

2.2.3 Future Reference To Shared Objects

Granting access by AID necessarily assumes that the server applet that wishes to share the object has foreknowledge of all AIDs of all client applets that are to be loaded which will share particular interfaces. This is a difficult requirement for any kind of server. But what about other client applets that are written afterwards which legitimately need access to the shared object? Such applets are excluded from access to the object, since the server can only grant access based on the AID list that the server had when it was loaded. This necessitates rewriting and reissuing the server applet such that the new AIDs are included, which may pose an insurmountable hurdle when the server applet is already widely distributed.

2.2.4 Inability To Pass Object Parameters

The inability to pass object parameters between applets can make many tasks cumbersome. For example, supposes as a client needs to pass an object when invoking a method on a server applet so that the server can manipulate and return the object, as in encrypting a buffer. In this case, the object might contain a field specifying the method of encryption, a key array, and a data array. But as stated earlier, the firewall prevents the server applet from accessing this data, even if an object reference from the client is provided.

One might envision a work around to this problem where the server A gets a shared interface on an object in client B in order to read the object data as shown in Figure 4.

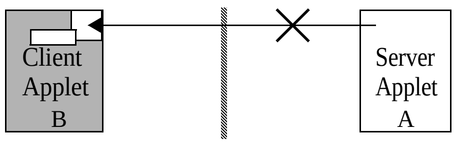


Figure 4: Server access blocked by firewall

However, this will fail, since any attempt by server A to get the object data from client B will also be blocked by the firewall. The server can only invoke methods on the client object, but the client object has no means for returning the object data, other than a single field at a time. Thus the only current work around to this problem is to use the global APDU buffer to pass data as shown in Figure 5.

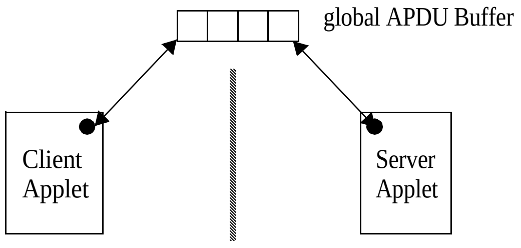


Figure 5: Data exchange via global APDU buffer

This is awkward at best, since the data being passed may not be of appropriate length or type to be readily passed via this mechanism. Consequently, the client and server might have to make multiple calls and considerable manipulations to pass the desired parameters, due to the restrictions of the APDU buffer.

This problem could be addressed by changing the firewall restrictions to permit access to shared objects (including data), instead of just interfaces; however, this could potentially create security holes. Applets must coded carefully to avoid exposing methods and data that are not intended to be shared. But if instead of actually sharing the applet object, the applet used a delegate to expose only the desired methods and data, the JCRE specification could be altered to permit access to only delegate methods and data, thus eliminating the object parameter problem, with a minimum of security risk.

3 Delegate Approach To Object Sharing

Based on an analysis of these critical problems, an approach to object sharing was devised which avoids these drawbacks. In this approach, each server applet that wishes to permit access to its methods or data creates a single delegate, which is registered with the system based on the AID of the applet. The delegate exposes only those methods and data that the applet wishes to share. Access to the delegate is public; all methods in the delegate can be accessed by any other applet!

Since access to the delegate is unrestricted, an applet protects itself in two ways. First, the delegate is written to only access the desired methods of the server applet; other applet methods cannot be accessed. Second, the methods of the delegate can perform any checks as deemed necessary to check the validity of the access to the delegate; if the checks are not passed, the delegate can refuse to pass the request on to the applet.

3.1 Java Card 2.1 System Changes Required

This approach presumes the addition of two new Java Card 2.1 system methods. A version of register is needed which takes a delegate and AID as arguments. A system method getDelegate returns the delegate associated with a particular AID. These methods replace the JCSystem.getAppletShareableInterfaceObject and Applet.getShareableInterfaceObject methods. Furthermore, the JCRE is altered to permit access to methods and data of delegate objects (DOs) in other contexts (just as it now permits access to methods of SIOs in different contexts).

For performance reasons, it would be worthwhile to investigate whether it is necessary for the JCRE to restrict access to objects in other contexts at all, since checking object context imposes a speed penalty. Techniques such as the delegate method proposed, coupled with classical Java language and runtime protections[11,5], could perhaps result in a better performing system that still meets the security requirements.

3.2 Delegate Server Implementation

A server applet must be written containing the desired methods. For this example, we show a loyalty server applet in Code 3 from the ABC company that allows granting, using, and reading loyalty points.

```
package AppABC;
import OSResources.*;

public class ABCLoyaltyApplet extends OSResources.Applet {
   private String aid = "AppABC.ABCLoyaltyApplet";
   private int loyalty;

   public ABCLoyaltyApplet() {
      loyalty = 0;
      register(new ABCLoyaltyDelegate(this), aid);
   }

   void grantPoints(int amount) {
         loyalty += amount;
   }

   boolean usePoints(int amount) {
      if (loyalty >= amount) {
         loyalty -= amount;
         return true;
      } else {
         return false;
      }
   }

   int readPoints() {
      return loyalty;
   }
}
```

Code 3: Server applet registering its delegate

Note that the only novel aspect of this applet is when it creates and registers its delegate, as illustrated in Figure 6, step 1. Whenever a server applet wishes to allow access to another applet, a delegate must be written which exposes the desired methods. Such a delegate is shown in Code 4.

```
package AppABC;
import OSResources.*;

public class ABCLoyaltyDelegate extends Delegate {

  final static byte CHALLENGE_LENGTH = (byte) 64;
  final static byte FAILURES_ALLOWED = (byte) 2;

  private byte[] secret1 = { /* initializing code ...*/ };
  private byte[] secret2 = { /* initializing code ...*/ };

   private ABCLoyaltyApplet myApplet;
   private ChallengePhrase cp;
   private ChallengePhrase checkcp;
   private byte grantTriesRemaining = FAILURES_ALLOWED;
   private byte useTriesRemaining = FAILURES_ALLOWED;

   protected ABCLoyaltyDelegate(ABCLoyaltyApplet a) {
      myApplet = a;
      cp = new ChallengePhrase(CHALLENGE_LENGTH);
      checkcp = new ChallengePhrase(CHALLENGE_LENGTH);
   }

   public boolean grantPoints(int amount) {
      // do some checking
      if (myApplet != null) {
         ABCLoyaltyInterface pD = (ABCLoyaltyInterface)
            OSystem.getDelegate(
               OSystem.getPreviousContextAID());
         if (pD != null) {
            /* Create new random challenge phrase */
            checkcp.setPhrase(cp.randomize());
            /* Get client response to phrase */
            byte[] response = pD.loyaltyChallenge(cp);
            byte[] r = checkcp.encrypt(secret1);
            if (isEqual(response,r)) {
               /* See if client gave the correct response */
               grantTriesRemaining = FAILURES_ALLOWED;
               myApplet.grantPoints(amount);
               return true;
            } else {
               if(--grantTriesRemaining == 0) myApplet = null;
               return false;
            }
         }
      }
      return false;
   }

   public boolean usePoints(int amount) {
   /* Access to this method uses secret2, for different
      clients, but is otherwise similiar to grantPoints ... */
   }

   public int readPoints() {
      if (myApplet != null)
         return myApplet.readPoints();
      else
         return 0;
   }
}
```

Code 4: Server delegate

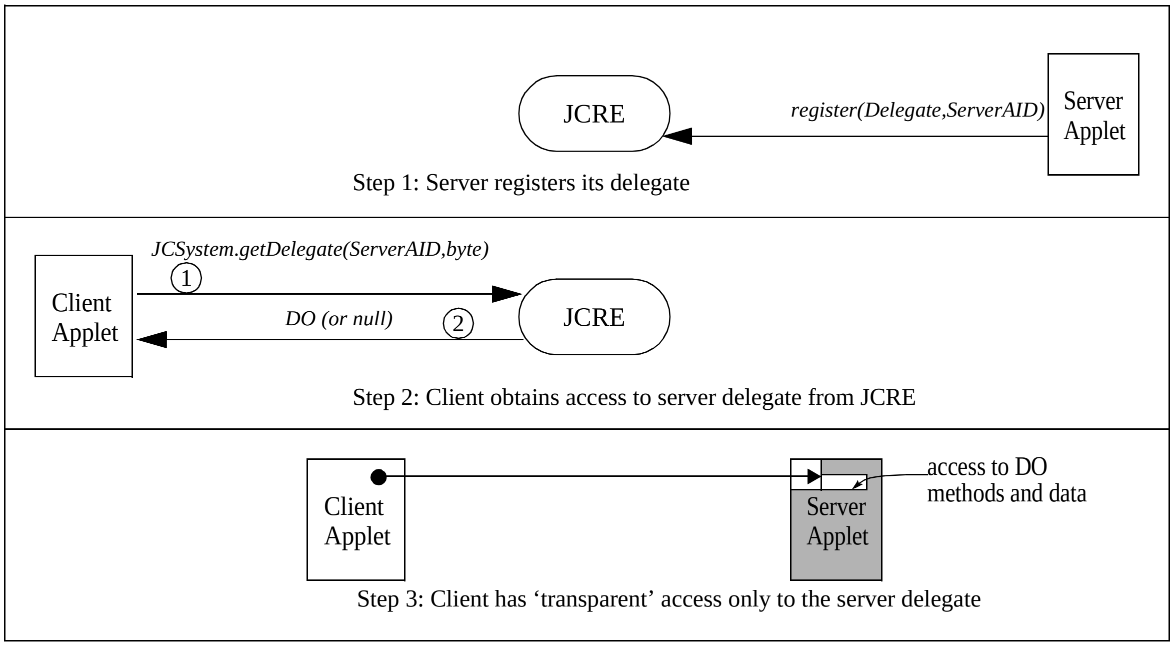

JCRE
register(Delegate,ServerAID)
Server Applet
Step 1: Server registers its delegate
JCSystem.getDelegate(ServerAID,byte)
1
Client Applet
DO (or null)
2
JCRE
Step 2: Client obtains access to server delegate from JCRE
Client Applet
Server Applet
access to DO methods and data
Step 3: Client has ‘transparent’ access only to the server delegate

Figure 6: Creating and accessing a delegate

This delegate is registered with the AID of the server applet at the time the applet is instantiated on the Java Card. At this point, this delegate is publicly accessible by any other applet. This particular delegate defines the interface ABCLoyaltyInterface shown in Code 5, which must be implemented by client applications. The reasons for this interface are discussed in detail in Section 3.3.

```
package AppABC;
import OSResources.*;

public interface ABCLoyaltyInterface{
   public byte[] loyaltyChallenge(ChallengePhrase cp);
}
```

Code 5: Server defines interface that each client must provide to handle challenge/response

3.3 Delegate Security

A delegate controls which methods each client can access. This delegate exposes three methods: grantPoints, usePoints, and readPoints. The method readPoints has no validation, and may be freely used by any client.

But what about the methods that require proof that the access is valid, such as grantPoints and usePoints? These methods could just check the AID of the client. This effectively mimics the object sharing of Java Card 2.1, which is subject to the risks of AID impersonation and difficulty of adding future clients.

These problems can be avoided by using the classical solution of shared secrets. In this example, access to grantPoints is controlled by secret1; only clients that can authenticate knowledge of secret1 will have their request passed from the delegate to the server applet. Access to usePoints is controlled by secret2, which allows a potentially different set of clients to access this method.

There are many standard techniques for handling shared secrets and other methods of authentication [3]. We are not proposing anything new here, other than the application of these techniques to this problem, to avoid the limitations and pitfalls of security based on AID. The details of the application of the shared secret technique to Java Card is presented in detail for the benefit of those who may not be familiar with such techniques, or may be unclear how they can be implemented in a Java card.

Proving knowledge of the secret could be a tricky proposition. Passing the secret as an argument to the server applet is a bad idea, since the secret could be intercepted in a number of ways. (One simple way to get the secret is to write a applet that impersonates the server applet, thus capturing the secret when it is passed as an argument.) A superior approach is to use challenge/response phrases, as illustrated by the server delegate in Code 4. When access is requested, the server delegate sends a random challenge phrase to a predefined method in the client delegate. This challenge phrase is encrypted using the secret by the client delegate, and returned to the server delegate. The server delegate performs the same encryption, and if the results match, access is granted. Thus the secret is never revealed outside of either applet.

3.4 Delegate Client Implementation

When a client applet wishes to use the server applet, it calls getDelegate with the AID of the server applet, and receives a delegate, which it casts to the proper delegate class associated with the server applet, as shown in Figure 6, step 2**.** The client applet then invokes methods on the delegate as desired. Note that unlike SIO, the JCRE hands out the delegate, and server applet is not involved. The client in Code 6 was written assuming authentication using challenge/response.

```
package AppPurse;

import OSResources.*;
import AppABC.*;

public class PurseApplet extends OSResources.Applet {

   public String aid = "AppPurse.PurseApplet";
   private byte[] secret1 = { /* initializing code ...*/ };
   private int value;

   public PurseApplet() {
      value = 0;
      register(new PurseDelegate(this), aid);
   }

   public byte[] loyaltyChallenge(ChallengePhrase cp) {
      return cp.encrypt(secret1);
   }

   public void use(int amount) {
      value -= amount;

      ABCLoyaltyDelegate d =
         (ABCLoyaltyDelegate)
         OSystem.getDelegate("AppABC.ABCLoyaltyApplet");
      if (d != null) {
         d.grantPoints(amount);
      }
   }
/* Other methods follow for adding points to the purse ... */
}
```

Code 6: Client applet invoking server method

To handle the challenge/response, the client must supply a delegate as shown in Code 7 that the server will call to perform the necessary encryption with the challenge phrase. This client delegate must implement the ABCLoyaltyInterface, as defined by the server applet. The delegate may not actually perform the encryption, but may instead pass the challenge/response request to the client applet for processing. This allows the delegate to avoid holding the shared secret, reducing the security risk.

```
package AppPurse;

import OSResources.*;
import AppABC.*;

public class PurseDelegate extends Delegate implements
ABCLoyaltyInterface{

   private PurseApplet myApplet;

   protected PurseDelegate(PurseApplet a) {
      myApplet = a;
   }

   public byte[] loyaltyChallenge(ChallengePhrase cp) {
      if (myApplet != null)
         return(myApplet.loyaltyChallenge(cp));
      else
         return null;
   }
}
```

Code 7: Client challenge/response delegate

3.5 Overview Of Client/Server Communication

At this point, it is presumed that the server has already registered with the JCRE, and that the client has already obtained the server delegate, as shown in Figure 6, and the client is ready to use a service from the server. The resulting client/server communication using a shared secret is illustrated in Figure 7.

The client begins by requesting a service from the server. This is done by invoking a method on the server delegate object ①. The method in the server delegate determines what level of protection is required; in this case, it determines that a shared secret must be validated to use this method. It therefore requests the delegate for the client from the JCRE ②, which the JCRE provides (if it exists) ③. Assuming the client delegate was successfully obtained, the server obtains a random challenge phrase, and sends it to the client delegate ④ using the interface the server had predefined for this purpose. The client delegate passes the challenge to the client applet (which contains the secret) ⑤. The client applet encrypts the challenge phrase using the shared secret, and returns the response phrase to the client delegate ⑥. The client delegate then passes the response phrase to the server delegate ⑦. The server delegate also encrypts the challenge phrase with the shared secret, and if the results match, the secret is validated. The server delegate then forwards the service request to the server applet ⑧, which processes the request, and returns a response to the delegate ⑨, which is forwarded to the client applet ⑩.

If the nature of the server applet is such that the services are likely to be reused by a given client in a single session, then after validation of the shared secret, the server would likely be designed to return a session key (which permits access only during the current session, which ends when power is cycled). The client could then access that particular service through the delegate by providing the session key, thus avoiding the overhead of validating the shared secret for each access of a delegate method. Although the session key is provided by the client to the server in clear text form, there is no real security risk, since the server randomizes the currently active session key each time power is cycled. So even if the session key were intercepted by an applet impersonating the server, it could not be used to breach security, due to the randomizing of the session keys.

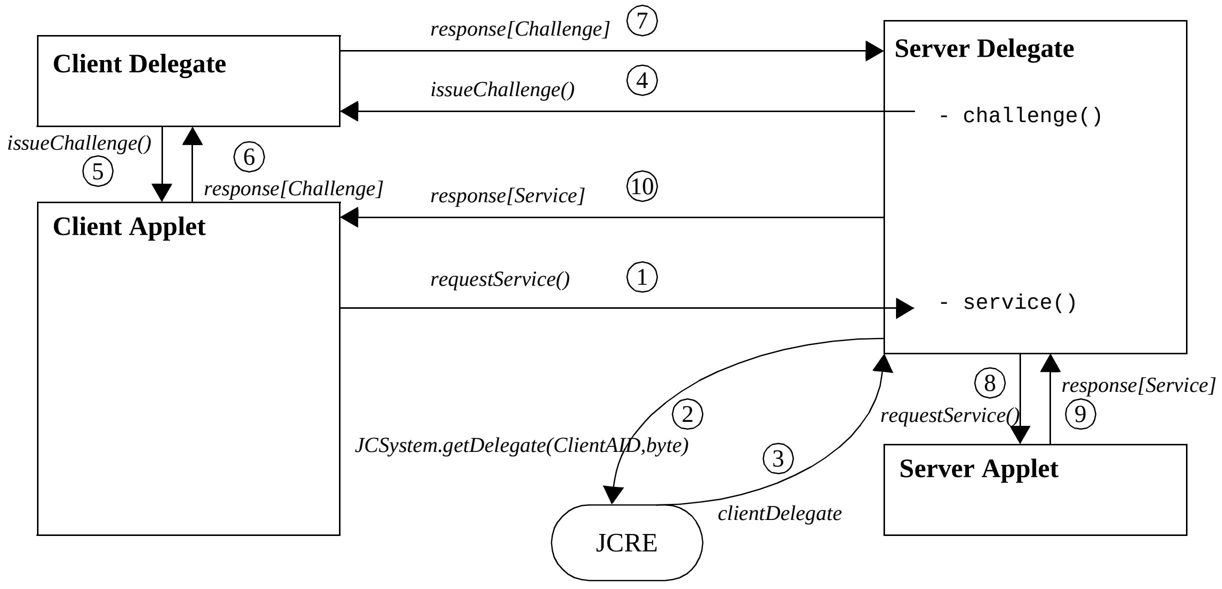


Figure 7: Challenge/Response validation of shared secret

4 Conclusion

The proposed delegate method of object sharing improves on the SIO approach. It protects each method as desired, so that no client applet gains access to unintended methods of a server applet. It avoids AID impersonation, since AIDs can be supplemented with shared secrets or other authentication mechanisms. It avoids future reference problems, since access need not be linked to any particular AID, but can simply be based on a shared secret which can easily be added to future client applications. Moreover, it requires only a minimal addition to the Java Card system. Since the delegate method allows each applet to determine the security policy desired on a per method basis, this allows the maximum flexibility for granularity of access by each individual client.

5 References

[1] Arnold, K. and Gosling, J., *The Java Programming Language*, Addison-Wesley, 1996.

[2] Guthery, Scott. B., *Java Card: Internet Computing On A Smart Card*, IEEE Internet Computing, pp. 57-59, Jan/Feb 1997.

[3] Guthery, Scott B., and Jurgensen, Timothy M., *Smart Card Developer's Kit*, Macmillian Technical Publishing, 1998.

[4] ISIO-7816, *Information Technology - Identification cards - integrated circuit cards with contacts.*

[5] McGraw, Gary E. and Felten, Edward W., *Java Security: Hostile Applets, Holes, and Antidotes*, John Wiley and Sons, 1996.

[6] McManis, Chuck. *My ENIGMatic Java Ring*. JavaWorld, 3(8), August 1998. http://www.javaworld.com/javaworld/jw-08-1998/jw-08-in-depth.html

[7] Sun Microsystems Inc., *Java Card 2.1 Virtual Machine Specification*, //java.sun.com/products/javacard/JCVMSpec.pdf

[8] Sun Microsystems Inc., *Java Card 2.1 Runtime Environment Specification*, //java.sun.com/products/javacard/JCRESpec.pdf

[9] Sun Microsystems Inc., *Java Card 2.1 Application Programming Interfaces Specification*, //java.sun.com/products/javacard/htmldoc/index.html

[10] Sun Microsystems Inc., *The Java Card 2.1 Platform Specifications*, //java.sun.com/products/javacard/

[11] Yellin, Frank., *Low level security in Java*, Fourth International World Wide Web Conference, Boston, MA, December 1995. http://www.w3.org/pub/Conferences/WWW4/Papers/197/40.html