Delegation

MAO-DOC-HOW-001 v1.1

Contents

How to Delegate

MULTOS applications reside within rigorously enforced firewalls. This means that an application can not directly access data or call functions held in another. There are situations, however, where applications need to share data or functions. MULTOS provides the mechanism of delegation to make this sharing possible.

How it Works – High level
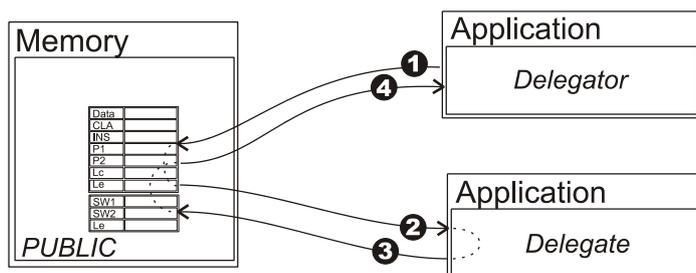
An application that makes use of delegation is referred to as the delegator. This application receives a command from the terminal and processes it normally. Prior to invoking delegation the delegator creates a command APDU in public (Arrow 1 in the figure). The APDU can be any ISO case.

When delegation is invoked the application to which the command is directed (known as the delegate) starts and looks to public memory to see what command needs processing (Arrow 2). The command is processed normally and the response including any data is placed in public memory (Arrow 3).

When the delegate exits MULTOS returns control to the delegator which will then examine the response from the delegate (Arrow 4) and proceeds accordingly.

Sample Implementation

Let's assume that there are two applications on the card. The first is an application that manages cardholder data and the second holds the cardholder PIN and can perform PIN verification. The first application will delegate any incoming PIN check command to the PIN checker application.

Building the APDU

The cardholder data management application uses a proprietary format for the PIN check APDU and data, where only the 2-byte BCD PIN value is sent as data. The PIN checker application uses an EMV plaintext PIN block and the command VERIFY. The data management application, the delegator, will need to change the incoming data to meet the delegate's criteria.

Sample C Code

The following code snippet defines the various lengths, structures and data used. It is based on SmartDeck C.

```c
// ***** Constant Definitions *****
#define L_PIN_BLOCK     8
#define L_PIN           2
#define L_BLOCK_PAD     5
#define L_DEL_AID       6
#define V_CLA           0x80
#define V_INS           0x20
#define V_P1P2          0x0000

typedef struct {
      unsigned char blockStart[1];
      unsigned char blockPIN[L_PIN];
      unsigned char blockPad[L_BLOCK_PAD];
} struc_PinBlock;

// ***** Data Definitions *****
// data arriving in public
#pragma melpublic
union {
         unsigned char pMyPIN[L_PIN];
         struc_PinBlock pDelegatePIN;
      } pdata;

#pragma melstatic
// For delegation to work the first byte of the AID
// memory area MUST indicate the length of the AID
unsigned char delegateAID[L_DEL_AID] = {0x05, 0xF0, 0x00, 0x00, 0x00, 0x01};

void main (void)
{
      // construct PIN Block
      memmove(pdata.pDelegatePIN.blockPIN, pdata.pMyPIN, L_PIN);
      memset(pdata.pDelegatePIN.blockStart,0x24, 1);
      memset(pdata.pDelegatePIN.blockPad, 0xFF, L_BLOCK_PAD);

      // construct APDU
      CLA = V_CLA;
      INS = V_INS;
      P1P2 = V_P1P2;
      Lc = L_PIN_BLOCK;

      // code continues…
```

Sample MULTOS Assembly Language Code

An APDU can be built using the MULTOS Assembly Language as well.

```
// ***** Constant Definitions *****
L_PIN_BLOCK  EQU  8
L_PIN        EQU  2
L_BLOCK_PAD  EQU  5
L_DEL_AID    EQU  5
V_CLA        EQU  0x80
V_INS        EQU  0x20
V_PARAM      EQU  0x00
START_BLOCK  EQU  0x24

// ***** Define APDU locations *****
pCLA         EQU    PT[-13]
pINS         EQU    PT[-12]
pP1          EQU    PT[-11]
pP2          EQU    PT[-10]
pLc          EQU    PT[-8]

// ***** Define Primitives Numbers *****
MemCpy       EQU    0x0C

// ***** Set Delegate Application AID *****
delegateAID STATIC BYTE 6 = 0x05, 0xF0, 0x00, 0x00, 0x00, 0x01

MAIN::

// copy incoming PIN to new location
// length, destination, source on stack
PUSHW        L_PIN
LOADA        PB[1]
LOADA        PB[0]
// invoke memory copy
PRIM         MemCpy

// populate rest of structure
SETB         PB[0], START_BLOCK
// load pad size bytes on stack
PUSHZ        L_BLOCK_PAD
// invert '00' to 'FF' on stack
NOTN         , L_BLOCK_PAD
// copy to public memory
PUSHW        L_BLOCK_PAD
LOADA        PB[2]
LOADA        DB[0]
PRIM         MemCpy

// Build the APDU
SETB  pCLA, V_CLA
SETB  pINS  V_INS
SETB  pP1   V_PARAM
SETB  pP2   V_PARAM
SETW  pLc   L_PIN_BLOCK

//the code continues…
```

Invoking Delegation

Once the APDU and accompanying data is ready the delegator calls the delegate application. The following code snippets continue from the previous examples.

In C the code would be:

```
Delegate(delegateAID);
```

In assembler it would be:

```
// name delegate primitive number
Delegate    EQU   0x80

// place address of delegate AID on stack
LOADA       delegateAID
// invoke the delegate primitive
PRIM        Delegate
```

What the Delegate Application Does

The delegate application looks to public and processes the APDU it finds there. In some cases, the application may wish to know if it has been delegated to and by which application. To do so the delegate can use the primitive "Get Delegator AID", which returns the AID of the delegating application or zero if the application is not a delegate. The delegate application can, then, decide if processing the command is permissible.

In our example the delegate application would check that the components of the VERIFY command were correct and, if so, would perform a PIN check. This processing is no different to that carried out when the PIN check application is selected directly. If an error is encountered at any point, the delegate sets the status word response appropriately and exits. In the case of a successful PIN check the status word is set to '9000' and the application exits. Control is then returned to the delegator.

Return from Delegation

When the delegate exits control is returned to the delegator application. At this point the delegator will most likely want to know the outcome of the delegated command. This can be done by interrogating the status word returned to see if it is '9000'.

In C it would be an if statement

```
if (SW12 != 0x9000)
{
    // send error message
}

// continue processing
```

In assembler a compare word would suffice:

```
// define SW location
pSW1        EQU   PT[-2]
pSW2        EQU   PT[-1]

// load sw onto stack
LOAD        pSW1, 2
```

```
CMPW        , 0x9000
// if not equal jump to exit label
BNE         _exit
// otherwise continue processing
```

What Happens in the Chip

The previous sections describe what delegation is and how to use it in code. It does not explain how the chip behaves during delegation. When the delegator invokes the primitive "Delegate" the operating system:

- caches the delegator application stack and session data
- activates the delegate application

After the delegate exits the operating system:

- caches delegate application stack and session data
- reactivates the delegator application

The fact that the delegate maintains its stack and session data means that it maintains it state. This allows a multiple command-response dialogue between delegator and delegate.

Things to look out for

There are a few things to be on the look out for. The first is linked to the primitive Checkcase, the second to potential status word ambiguity and a third to how stack and session data caching is implemented.

Checkcase for ISO Cases 3 and 4

The primitive Checkcase instructs the OS how to interpret incoming APDU. An ISO case 3 or case 4 command contains command data. Under T=0 when Checkcase is called with either of these values the OS prompts the terminal for data of length Lc.

It is very likely that the delegator has used Checkcase and received all available data. If the delegate were to use Checkcase again the terminal would be prompted to send data, but it would have no data to send. The result would be a time out and aborted processing. There are a few ways to work around this and which to use depends on how much control the developer has over both applications.

One work around is to avoid performing Checkcase in the delegator and have the delegate do it always. This would be appropriate if there is little or no control over the delegate application and the data is to be passed as is.

Another work around would be appropriate where the developer can update the code in the delegate application. When Checkcase is called and the command data is received MULTOS sets the bit flag CmdDataRxd, found in bit 3, in the Protocol Flags byte found at PT[-17]. The delegate application can interrogate the bit flag and skip using Checkcase.

The C code for interrogating that flag would be:

```
Unsigned char result[1];
Unsigned char operand[1]= {0x04};
```

```
// operand used in AND to see if bit 3 is set
// result of AND written to result byte
ANDN(1, result, operand, ProtocolFlags)

// interrogate result byte
// cast as int for ease of comparison
if ((int)result != 0)
{
     // CmdDataRxd not set
     // therefore, do checkcase
     CheckCase(2);
}

// otherwise skip and continue with code
```

The assembler would be:

```
PUSHB       0x04        // comparison value on stack
LOAD        PT[-17], 1  // Protocol Flag value on stack
ANDN        , 1         // AND two stack bytes
POPW                    // remove bytes from stack
JNE         _SkipCheckCase  // JNE as result of AND != 0
```

Ambiguous Status Word

An application can use delegation, but there are cases where it might fail. The causes of delegation failure are:

- There is no application with the specified AID on the card
- The AID specified is not between 1 and 16 bytes inclusive in length
- The delegate application is active; i.e., recursive delegation is not permitted
- The implementation defined maximum number of delegations has been reached

In all of these cases delegation will fail and MULTOS will set the Status Word to '6A 83', which is defined in ISO as "record not found".

The possibility of ambiguity arises when the delegate application uses that same status word for an application specific error condition. The delegator will call the primitive delegate and when control is returned it will check the status word returned. If it were to be '6A 83', the delegator would be unable to tell if that meant delegation failed or that the delegate application is reporting an error condition.

In the case where a failure of any sort ends processing this does not pose a problem as the delegator would simply report the error to the terminal. However, this does not apply if specific action is required in the case of delegate error.

Stack and Session Data

The contents of dynamic are preserved during a delegation. After the delegate has completed, the delegator continues execution at the instruction after the delegate call, and with session data in the same state.

The contents of the delegate's session data is also preserved after it has exited. If the same application is used as a delegate again, then its dynamic area will be preserved between delegations.

If a new file is selected by the IFD, i.e. the current session ends, then all dynamic data is cleared. During a session, all dynamic data is preserved for all applications which are executed.

----- End of Document -----