Security-by-Contract for Open Multi-Application Smart Cards

O. Gadyatskaya, F. Massacci (University of Trento)
B. Chetali, Q.-H. Nguyen (Trusted Labs, Gemalto)

e-Smart’2011
September 21-23, Sophia-Antipolis
Plan of the talk

• Motivations
• Java Card
• The Security-by-Contract solution
• Technical obstacles for prototype implementation
• The implementation highlights
• Conclusions
Plan of the talk

• Motivations
• Java Card
• The Security-by-Contract solution
• Technical obstacles for prototype implementation
• The implementation highlights
• Conclusions
Open multi-application cards

picture from http://fingerprints-security.net

"One card to rule them all"

Olga Gadyatskaya
01/2016
Open multi-application cards II

• One card can host multiple applications
• Applications are coming from different providers
• Applications can be installed or removed
• Applications can interact on the card to provide a given service

How to ensure their interactions are authorized?
The challenge

The card has to verify that the policies of all applets are satisfied

• set of applications to run on card is unknown initially
• evolution occurs unexpectedly
• each application has its own policy on interactions
• approach must work for a smart card:
  • run-time monitoring is not possible
  • algorithms have to be small and fast
An example

Olga Gadyatskaya
01/2016

ePurse App

University App

Trento Bus App

Shop Loyalty App

1
An example

How to ensure that new application will interact only with applications that allowed this interaction?
Plan of the talk

• Motivations
• Java Card
• The Security-by-Contract solution
• Technical obstacles for prototype implementation
• The implementation highlights
• Conclusions
Loading process on Java Card 2.x.x

- Compiler
  - .java
  - .class
- Converter
  - .java
  - .class
  - CAP file
- JCRE
  - Installer
  - Instance
  - Applet A
  - Firewall
  - Applet B
- Java Card API
- Native API
- Native OS
- Integrated Circuit
- .java
- .class
- Compiler
- Converter
- CAP file
Application interactions on Java Card

Run-time

Applet A
Shareable Interface
  service 1
  service 2

Firewall

Applet B
Why the current architecture does not work?

• Once an application is installed, it can do whatever (try to call anybody)
• So the callee must check who is calling
• The execution logic of an applet is currently interleaved with the access control logic
• This is
  – not flexible
  – error-prone
Plan of the talk

• Motivations
• Java Card
• The Security-by-Contract solution
• Technical obstacles for prototype implementation
• The implementation highlights
• Conclusions
The SxC Workflow

1. **Loading**
   - CAP
   - Bytecode
   - Contract

2. **Claim Checker**
   - Contract matches Bytecode?

3. **Policy Checker**
   - Contract matches Policy?
     - Yes
     - No

4. **Policy Storage**
   - Policy App 1
   - ... Policy App N

5. **Retrieve Policy**

6. **Update Policy**

7. **Linking and Installation**

8. **Stop**
   - Free the memory

9. **Integrated with the Installer**

10. **Yes**

11. **No**
## Contracts for applet interactions

### Contract of a package

<table>
<thead>
<tr>
<th>AppClaim</th>
<th>AppPolicy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Provided services</strong>&lt;br&gt; &lt;Interface token, method token&gt;</td>
<td><strong>Authorizations for services access</strong>&lt;br&gt; &lt;Interface token, method token, Authorized package AID&gt;</td>
</tr>
<tr>
<td><strong>Called services</strong>&lt;br&gt; &lt;Provider package AID, Interface token, method token&gt;</td>
<td><strong>Functionally necessary services</strong>&lt;br&gt; &lt;Provider package AID, Interface token, method token&gt;</td>
</tr>
</tbody>
</table>
Example of a contract

A simple applet: 1 provided service, which can be used by 1 applet; 1 called service

Contract of **CoopLoyaltyApplet**:

{ 0x0, 0x1, 0x0, 0x0, 0x0, 0x1, 0x0, 0x0, 0x0, 0x1, 0x0, 0x0, 0x6, 0x1, 0x2, 0x3, 0x4, 0x5, 0x0, 0x0, 0x01, 0x8, 0x1, 0x2, 0x3, 0x4, 0x5, 0x6, 0x7, 0x0, 0x0, 0x0, 0x1, 0x0, 0x0 }  

Totally 31 byte
Security policy of the card

Collection of all loaded applets’ contracts

- ePurse App
- University App
- Shop Loyalty App 1
- Shop Loyalty App 2

Policy
Olga Gadyatskaya
01/2016
Plan of the talk

• Motivations
• Java Card
• The Security-by-Contract solution
• Technical obstacles for prototype implementation
• The implementation highlights
• Conclusions
Implementation of the SxC framework

• We experimented the SxC approach by implementing the framework as a proof-of-concept prototype that could be integrated with an industrial card

• The requirements for the prototype:
  – The framework needs to be written in C
  – Full memory footprint of the SxC prototype occupies up to 30 KB
Technical obstacles we have faced

1. Policy Storage requires access to EEPROM
2. Policy Storage organization needs to be efficient
3. CAP file parsing needs to be efficient
4. Contract delivery needs to be secure

- Policy Storage
  - Policy App1
  - ... Policy App N

- Claim Checker
  - Contract matches Bytecode?

- Policy Checker
  - Contract matches Policy?

- Update Policy
- Linking and Installation

- Stop
- Free the memory
Obstacle 1: EEPROM for the Policy storage

• Only modifiable persistent memory can be used to store the policy after each evolution
• Only applet instances are entitled an access to the EEPROM
• The Claim Checker and the Policy Checker are implemented in C

Solution:

• We implemented the Policy storage as the Policy Applet.
• The communication between the Policy Applet and the C components is implemented through the APDU buffer (a temporary solution)
Obstacle 2: Policy storage optimization

• AIDs are space-consuming objects
• The algorithms need to be fast

Solution:

• We organized the Policy storage efficiently using a fixed Policy structure and bit-vectors
Security Policy on the card

We assume 4 loaded applets, 8 services each

---

**Policy on the card**

- **Policy (fixed size)**
  All loaded contracts in an internal bit-arrays format

- **Mapping**
  Maintains correspondence between on-card ID and AIDs

- **MayCall**
  Possible future authorizations for applets not yet on the card

- **WishList**
  Called services from applets not yet on the card

---

Small size and (frequent) efficient operations

Big size and (rare) slow operations
Obstacle 3: contract delivery

• Contracts need to be embedded in CAP files

Solution:

• We embed contracts into Custom components
• Alternative solution would be to use the Static Field component
Obstacle 4: CAP file parsing efficiency

• Preferably each component should be parsed only once

• RAM usage optimization is necessary

Solution:

• We use specific 255 bytes temporary buffer for storing the computation data
The SxC architecture for loading

- .java
- .class
- Compiler
- Converter
- CAP file with contract
- Contract
- .class
- Policy Applet
- Applet A
- Firewall
- Applet B
- JCRE
  - SxC
    - Claim Checker
    - Policy Checker
  - Installer
  - Java Card API
    - JCVM (Interpreter)
    - Native API
  - Native OS
  - Integrated Circuit
- Compiler
- Converter
- CAP file with contract
- Contract
- .java
- .class
- Policy Applet
- Applet A
- Firewall
- Applet B
- JCRE
  - SxC
    - Claim Checker
    - Policy Checker
  - Installer
  - Java Card API
    - JCVM (Interpreter)
    - Native API
  - Native OS
  - Integrated Circuit

O. Gadyatskaya  The SxC Framework for Java Card
Plan of the talk

• Motivations
• Java Card
• The Security-by-Contract solution
• Technical obstacles for prototype implementation
• The implementation highlights
• Conclusions
The SxC prototype architecture implemented on a PC simulator

Java Stub for plugging Java components

Policy Checker

SxCInstaller

Claim Checker

CAP file parsing library contains APIs to access each CAP file component

Java Card

Policy Applet
Memory footprint of the prototype

<table>
<thead>
<tr>
<th>Component</th>
<th>Memory (bytes)</th>
<th>LOCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>SxCInstaller</td>
<td>6754 B</td>
<td>152</td>
</tr>
<tr>
<td>Claim Checker</td>
<td>6522 B</td>
<td>162</td>
</tr>
<tr>
<td>Policy Applet</td>
<td>2282 B</td>
<td>101</td>
</tr>
</tbody>
</table>

- Occupied EEPROM
- Object files
Plan of the talk

• Motivations
• Java Card
• The Security-by-Contract solution
• Technical obstacles for prototype implementation
• The implementation highlights
• Conclusions
Industry-Academia validation

Or How to validate the framework while protecting the smart card platform APIs

• The prototype relies on the CAP file parsing library
• Thorough industrial testing with actual CAP file parsing APIs is being done
Future work

How the card can manage possible conflicts among applications?

What if ePurse App wants to be removed and Trento Bus App relies on it?

Alternatives:
- ePurse is forced to stay
- Trento Bus is disabled
Thank You!

contact us at olga.gadyatskaya@unitn.it for more information

or visit http://dipi.unitn.it/~gadyatskaya/

www.securechange.eu