DD2460 Software Safety and Security: Part III
Laboratory: Jif

Gurvan Le Guernic

DD2460 (III, L1)
March 13th, 2012

Warning: many of the problems you will encounter in this lab assignment have already been discussed in the Jif exercises session (February 22nd). It is strongly advised to have done and understood the Jif exercises session before starting this lab.

1 Description of lab project

In this lab, you will program a model of a credit card payment protocol for the auction system. Five entities interact in this protocol:

- the `Auctioneer` drives the protocol
- the `Buyer` won the bet and has to pay the winning amount
- the `Seller` is to receive the money paid by the `Buyer`
- `Buyer's bank` handles money manipulation for the `Buyer`
- `Seller's bank` handles money manipulation for the `Seller`

Figure 1 depicts the different steps of the protocol when everything goes well. The main method driving the protocol is `AuctionSystem.processCCPayment` which proceeds as follows:

1. retrieve the bank and accountId of the winner/buyer and vendor/seller;
2. call `requestPayment` on the buyer's bank and store the result into `authorizationCode`;
3. call `validateAuthorizationCode` on the seller's bank with the previously retrieved `authorizationCode`, and store the result into `pymtOk`;
4. if `pymtOk` is false:
   (a) call `sendMessage("Payment failed")` on the buyer and seller;
   (b) exit by returning `false`.
5. call `sendMessage("Payment validated (" + amount + ")")` on the buyer;
6. call `validateTransaction(amount)` on the seller, and store the result in `sellersValidation`;
7. if `sellersValidation` is false:
   (a) call `sendMessage("Transaction canceled by seller.")` on the buyer;
   (b) call `cancelPayment(authorizationCode)` on the buyer's bank, and store the result in `canceled`;

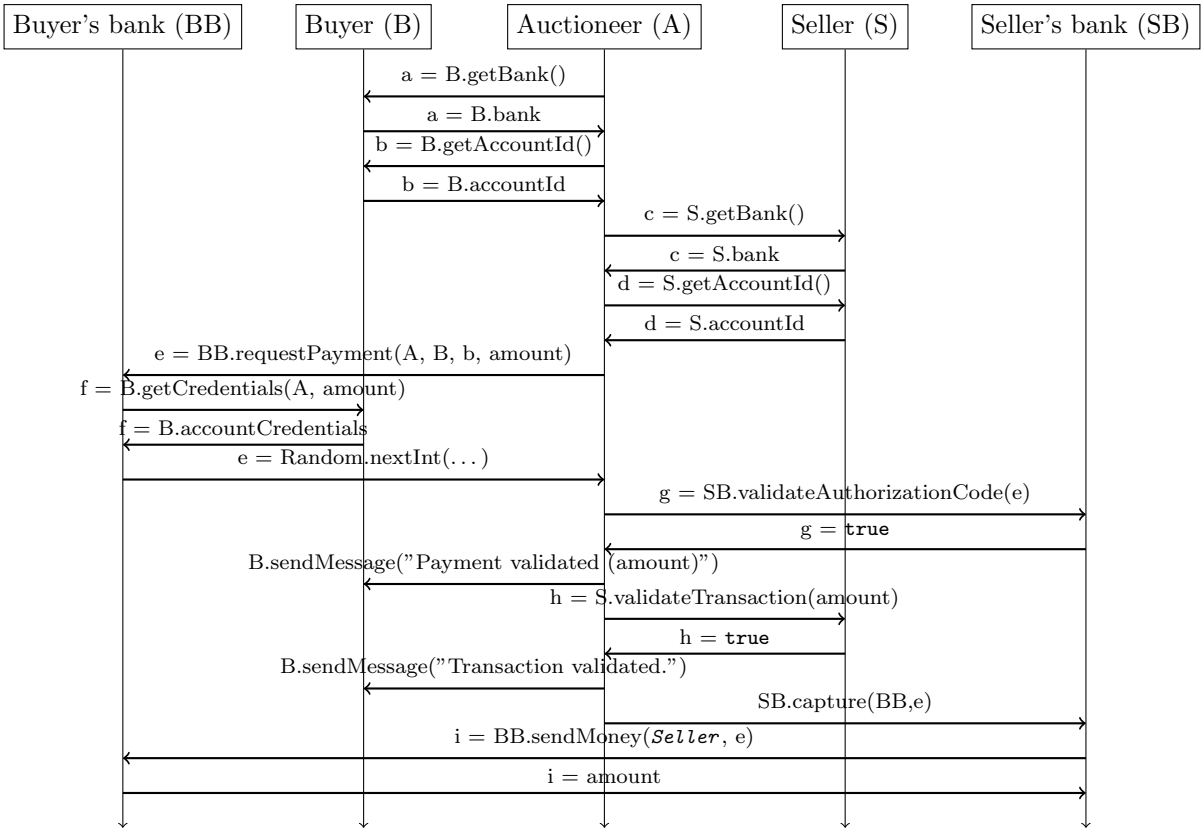

Figure 1: Credit card payment protocol

(c) if `canceled` is true then call `sendMessage("Payment has been canceled.")` on the buyer, otherwise call `sendMessage("Problem canceling payment. Contact your bank")` on the buyer;

(d) exit by returning `false`.

8. call `sendMessage("Transaction validated.")` on the buyer;
9. call `capture(buyersBank, authorizationCode)` on the seller's bank;
10. return `true`.

`User` and `Bank` methods are described in their respective source files.

2 Setting up the lab environment

Before starting, download from the labs webpage the file containing the program skeleton and unzip it. The skeleton is composed of the following files:

- `setup-lab_payment.sh`: a script to finish initializing the lab environment. **Run it now.**
- `jifc.sh`: the script to use to compile your project (**do not use the default jifc command**)
- `jif.sh`: the script to use to run your program
- `sig-src/*`: additional Jif signatures for some java methods you will need (Random.nextInt, HashMap.put and HashMap.remove) (**do not modify**; unless you need signatures for additional java classes and/or methods and you know what you are doing)

- `sig-classes/*`: files generated when `setup-lab_payment.sh` compiles files in `sig-src` (**do not modify**)
- `jif-src/jif/*`: the Jif definition of the 3 principals (*`Auctioneer`*, *`Seller`* and *`Buyer`*) we use for this project (**do not modify**)
- `jif-classes/jif/*`: files generated when `setup-lab_payment.sh` compiles files in `jif-src/jif` (**do not modify**)
- `jif-src/auctionPayment/*`: the Jif code of the credit card payment protocol
- `jif-classes/auctionPayment/*`: files generated when compiling files in `jif-src/auctionPayment` (**do not modify**)

The code of the protocol (in `jif-src/auctionPayment`) is composed of the following files:

1. `Main.jif` (**do not modify**) contains the main method used to test the protocol implementation.
2. `PubliclyNamedEntity.jif` (**do not modify**) define a public field common to any named entity. It also contains 2 helper input/output methods that you will need to use:
   - `output(String str)` outputs the message `str` to `this`. Use exclusively this output statement; do not use `Main.println(...)`.
   - `input()` reads (inputs) a line provided by `this`. Use exclusively this input statement; do not use `Main.readLine(...)`.
3. `AuctionSystem.jif` (should) contain the definition of the payment protocol.
4. `User.jif` (should) contain the definitions of the methods needed to interact with users.
5. `Bank.jif` (should) contain the definitions of the methods needed to interact with banks.

To run the Jif compiler: `./jifc.sh -e jif-src/auctionPayment/Main.jif`

To run your program: `./jif.sh auctionPayment.Main`

2.1 Stating the information flow policies

In this project, we only use 3 principals (*`Auctioneer`*, *`Seller`* and *`Buyer`*) to state the DLM labels we need. However, the code is generic. Classes `User` and `Bank` (as well as `PubliclyNamedEntity` but you do not have to modify it) are parametrized with a principal parameter *`P`* (that in this project will stand for *`Seller`* or *`Buyer`* in all the instances of `User` and `Bank` that we use in this project).

All the needed principals (*`Auctioneer`*, *`Seller`* and *`Buyer`*) are already defined in `jif-src/jif/principals` and automatically compiled when running `setup-lab_payment.sh`. You can have a look into the principals directory if you want, but you should not modify it.

To be done: Add the necessary labels to state the following confidentiality policies (we do not handle integrity policies). **Be explicit** in your label definitions, state all the readers ($\{P \rightarrow\}$ is different from $\{P \rightarrow P\}$; even if $\{P \rightarrow P\} \leq \{P \rightarrow\}$).

1. `User[P].bank` is readable by anybody. There is no confidentiality requirement.
2. `User[P].accountId` is readable only by *`P`*. A policy controlled by (i.e. under the responsability/ownership of) *`P`* (and anybody that can act for *`P`*) states that the only allowed reader is *`P`*.
3. `User[P].accountCredentials` is readable only by *`P`*. A policy controlled by (i.e. under the responsability/ownership of) *`P`* (and anybody that can act for *`P`*) states that the only allowed reader is *`P`*.

4. `User[P].getBank()` returns a value readable by anybody. Nobody states any requirement.

5. `User[P].getAccountId()` returns a value readable only by *P*. A policy controlled by (i.e. under the responsability/ownership of) *P* (and anybody that can act for *P*) states that the only allowed reader is *P*.

6. `User[P].getCredentials()` returns a value readable only by *P*. A policy controlled by (i.e. under the responsability/ownership of) *P* (and anybody that can act for *P*) states that the only allowed reader is *P*.

7. `User[P].validateTransaction(int)` returns a value readable only by *Auctioneer*. A policy controlled by (i.e. under the responsability/ownership of) *Auctioneer* (and anybody that can act for *Auctioneer*) states that the only allowed reader is *Auctioneer*.

8. `Bank[P].nextAccountId` is readable only by *P*. A policy controlled by (i.e. under the responsability/ownership of) *P* (and anybody that can act for *P*) states that the only allowed reader is *P*.

9. `Bank[P].accountId` is readable only by *P*. A policy controlled by (i.e. under the responsability/ownership of) *P* (and anybody that can act for *P*) states that the only allowed reader is *P*.

10. `Bank[P].accountCredentials` is readable only by *P*. A policy controlled by (i.e. under the responsability/ownership of) *P* (and anybody that can act for *P*) states that the only allowed reader is *P*.

11. `Bank[P].balance` is readable only by *P*. A policy controlled by (i.e. under the responsability/ownership of) *P* (and anybody that can act for *P*) states that the only allowed reader is *P*.

12. `Bank[P].onHold` is readable only by *P*. A policy controlled by (i.e. under the responsability/ownership of) *P* (and anybody that can act for *P*) states that the only allowed reader is *P*.

13. `Bank[P].holdRecords` is readable only by *P*. A policy controlled by (i.e. under the responsability/ownership of) *P* (and anybody that can act for *P*) states that the only allowed reader is *P*.

14. `Bank[P].setBalance(int)` returns an array whose structure (length, ...) is readable by anybody (There is no confidentiality requirement on the structure); but containing values that are readable only by *P* (A policy controlled by (i.e. under the responsability/ownership of) *P* (and anybody that can act for *P*) states that the only allowed reader is *P*). For information, the returned array contains the account ID and credentials chosen by the bank for the associated user. The returned value is used to initialize the related fields in `User[P]`.

15. `Bank[P].getBalance()` returns a value readable only by *P*. A policy controlled by (i.e. under the responsability/ownership of) *P* (and anybody that can act for *P*) states that the only allowed reader is *P*.

16. `Bank[P].getAmountOnHold()` returns a value readable only by *P*. A policy controlled by (i.e. under the responsability/ownership of) *P* (and anybody that can act for *P*) states that the only allowed reader is *P*.

17. `Bank[P].requestPayment(...)` returns a value readable only by *Buyer* and *Seller* under the control of *Auctioneer*. A policy controlled by (i.e. under the responsability/ownership of) *Auctioneer* (and anybody that can act for *Auctioneer*) states that the only allowed readers are *Buyer* and *Seller*.

18. `Bank[P].validateAuthorizationCode(...)` returns a value readable only by *Auctioneer*. A policy controlled by (i.e. under the responsability/ownership of) *Auctioneer* (and anybody that can act for *Auctioneer*) states that the only allowed reader is *Auctioneer*.

19. `Bank[P].cancelPayment(...)` returns a value readable only by *Auctioneer*. A policy controlled by (i.e. under the responsability/ownership of) *Auctioneer* (and anybody that can act for *Auctioneer*) states that the only allowed reader is *Auctioneer*.

20. `Bank[P].sendMoney(principal P2, ...)` returns a value readable only by *P2*. A policy controlled by (i.e. under the responsability/ownership of) *P2* (and anybody that can act for *P2*) states that the only allowed reader is *P2*. For information, the result of this method is supposed to be used to update the balance of a user that can act for *P2* (*Seller* in the case of this protocol).

Questions: Justify the security policies for:

- `User[P].validateTransaction(int)`,
- `Bank[P].setBalance(int)`
- `Bank[P].requestPayment(...)`,
- `Bank[P].validateAuthorizationCode(...)`,
- `Bank[P].cancelPayment(...)`,
- `Bank[P].sendMoney(principal P2, ...)`.

To justify those policies, you need to think about how the values returned by those methods are computed and how they will be used.

2.2 Coding a credit card payment method

To be done: Code the missing bodies of the methods in `AuctionSystem`, `User` and `Bank`.

Instruction: declassification statement `declassify` is dangerous (it usually breaks the security policy). It should be used only when there is no other way (you will need to use it a few times). **Every use of `declassify` has to be justified in the report**. Use only the "`declassify(`e`, `$fromLabel$` to `$toLabel$`)`" expression and "`declassify(`$fromLabel$` to `$toLabel$`)` S" statement (**do not use the shorter versions without** $fromLabel$).

Instruction: `NullPointerException`s should be handled by relying on a local null pointer analysis as described in exercise 2.9 from the second exercises session. Other exceptions should be handled by using "`try catch`" statements. No `throws` statements should be added to the method signatures.

Advices:

- make regular backups of your source code (it is easy to start screwing up your own code) (or use a version control mechanism);
- compile the files as often as possible to detect errors early;
- take the time to read and understand the error messages (they are usually surprisingly accurate; even if sometimes, to correct a compilation error at a specific line, you need to modify something a few lines above, or the signature of another method);
- comment all the methods that generate errors and are not used yet;
- code the protocol step by step:
  1. start by listing all the methods in `User` and `Bank` needed to code the first protocol step in `processCCPayment` (a = B.getBank());
  2. iterate the following until all methods in the previous list are coded: for every method in the list that do not need to call a method not coded yet, code it, compile it, and debug it;
  3. code the first protocol step, compile, debug, and test the project;

4. do the same for all other protocol steps.

- it is usually better/easier to have a single `return` statement per method.
- in the methods of the 2 generic classes (`User[P]` and `Bank[P]`) that are meaningful only for one principal (*`Seller`* or *`Buyer`*), such as `Bank[P].requestPayment` or `Bank[P].sendMoney` which are called only on `Bank[Buyer]` instances and never on `Bank[Seller]` instances, you *may* need to use "`actsfor`";
- you *may* need to add `where authority(...)` to the signature of methods which break the security policy;
- `PubliclyNamedEntity.output(...)` and `PubliclyNamedEntity.input()` may trigger compilation errors when called under some specific conditions (check their definitions);
- if you need to `print` something to debug your code, the statement most likely to compile (or easier to debug) is the following: "`declassify({}) this.output(declassify("...", {}))`";
- remember, sometimes, to correct a compilation error at a specific line, you need to modify something else somewhere else, maybe even in another class;
- try to understand why you use a specific label (what does it mean from a security point of view?); if it does not make sens, then it probably does not make sens;
- as a rule of thumb, ask yourself the following questions, especially when you use `declassify` and/or `authority`:
  - Can you justify your security labels ? Are they acceptable from a security point of view ? In any case, you will have to justify them in your report whenever you use `declassify` and/or `authority`.
  - If part of the information flow policy has to be broken, is it acceptable with regard to the overall security policy and who (among the three principals Buyer, Seller, Auctioneer) should have the right to break this specific part of the policy ?

Expected behavior: When running the project, you should get:

- if you answer "Y" and "Y"

```
*** Starting application ***

*** Entities initialized ***

Closing auction: Bob --(1000)--> Alice

Bob> Do you want to validate payment of 1000 requested by Kauction?
Bob> Y
Bob> Payment validated (1000)
Alice> Do you validate transaction for 1000?
Alice> Y
Bob> Transaction validated.

*** Application end ***

 Alice -> 11000 (- 0)
 Bob -> 9000 (- 0)
```

- if you answer "Y" and "N"

```
*** Starting application ***

*** Entities initialized ***

Closing auction: Bob --(1000)--> Alice

Bob> Do you want to validate payment of 1000 requested by Kauction?
Bob> Y
Bob> Payment validated (1000)
Alice> Do you validate transaction for 1000?
Alice> N
Bob> Transaction canceled by seller.
Bob> Payment has been canceled.

*** Application end ***

 Alice -> 10000 (- 0)
 Bob -> 10000 (- 0)
```

- if you answer "N"

```
*** Starting application ***

*** Entities initialized ***

Closing auction: Bob --(1000)--> Alice

Bob> Do you want to validate payment of 1000 requested by Kauction?
Bob> N
Bob> Payment failed
Alice> Payment failed

*** Application end ***

 Alice -> 10000 (- 0)
 Bob -> 10000 (- 0)
```

2.3 Coding a stealing method

For this exercise, **do not modify any method**; code new methods if needed.

To be done: Uncomment the last line in `Main.main`, code the required method. This method (`User[P].stealAccountId(principal V, User[V] victim)`) retrieves the accountId from `victim` and display it to the thief (not the victim) by using `this.output(...)`.

Questions: What do you need to do to code this method? Comment.

2.4 Content of the report

In you report, you should:

- answer to the "Question:" paragraphs,

- justify every use of the `authority(...)` annotation, in method and class signatures, and `declassify` statement/function in your code (why it is needed, why it is not possible to do without, why the modification of the security label is acceptable with regard to the overall functionality and security of the system),
- comment on the 3 most difficult methods from `User` and/or `Bank` (from a security labeling point of view) by explaining why it was difficult and how you solved this difficulty (inside the method and at call sites).