README.md

Ex 1 Concurrency theory - modelling with Spin

The task specification

We are given two programs - the encoder and the decoder.

- Encoder receives message `X` and public key `k_-1` and returns enrypted message `{X}k_-1`
- Decoder receives message `{X}k_-1` and key `l` and returns `X` only and only if `k == l`

We will be working with asymmetric protocol that looks like the following:

- Party `A` downloads from `CA` server public key `kB_-1` of party `B` that it wants to connect to
- `A` chooses random message `X` and sends message `{X, A}kB_-1` to `B`
- Party `B` receives that message, decodes, downloads from the `CA` the public key of `A` and sends the message `{X, A}kA_-1` back to `A`
- `A` receives that message and sends back `{X, Y}kA_-1` where `Y` is just a randomly chosen mesage
- `Y` is now a common secret shared by both of the parties

The task is to model the system in Promela and find the technique that can be exploited by the attacker to compromise the connection.

Initial model

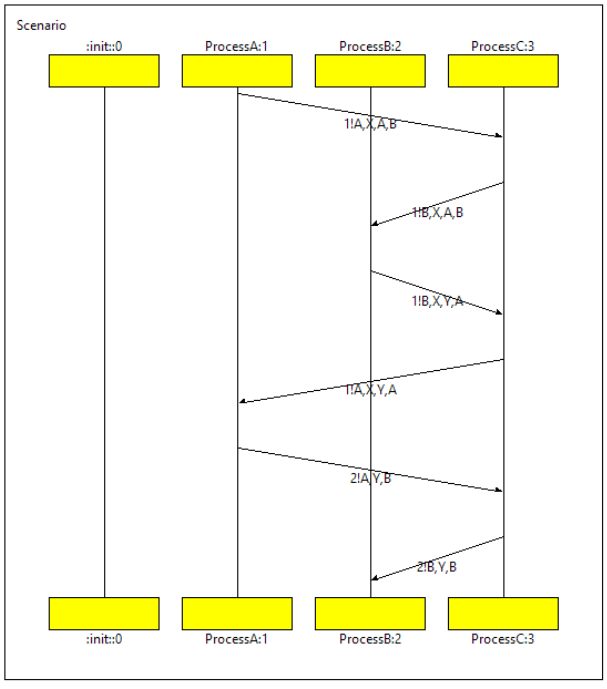
The first approach was to model the system such way that the nonces (X, Y) are radomly selected from range of predefined values and the message itself consist of 2-element array of bytes. Such model is hard to work with as we do not need to know exact values of nonces, but rather we care about the fact of knowing them. It also have a lot of flaws as it was the very first approach of mine to modeling in Pamela language.

The model is available in file prototype.pml

Improving the model

For simplicity we assume that for now all parts of key-exchange algorithm do not communicate with certificate provider (all public keys are known to everyone). That assumption clarifies all MSC diagrams and reduces trivial code.

The basic model with A, B and basic interceptor C (who passes messages transparently) is present in the file basic_model.pml



Code structure

The channels:

```
chan chExchange = [0] of {
        mtype:DataType, mtype:DataType, mtype:DataType, mtype:DataType
};

chan chConfirm = [0] of {
        mtype:DataType, mtype:DataType, mtype:DataType
};
```

Are defined as zero-capacity Rendezvous ports for passing 3 and 2-components messages.

The messages have the following format:

```
<source_of_message/destination> <data1> <data2> <public_key_used_for_encryption>
  representing {data1, data2} PUB(key)
      or
<source_of_message/destination> <data> <public_key_used_for_encryption>
  representing {data} PUB(key)
```

`source_of_message/destination` can be anyting (i.e. `A`, `B` or `C`), data can be `X`, `Y`, `ANY` (representing any data value - explained later)

For messages sent to channel we assume that the value represents the source of a message. For messages received from the channel we assume that the value represents the destination of a message.

Processes

The process C reads messages sent by A and B and switches first element of tuple thus forwards them to valid destinations. (process C is a transparent proxy).

Interceptor knowledge base

The model itself explicitly defines MITM attack model.

To make the Spin find the solution to the insecure protocol we have to specify all meaningful possible combinations of inputs/outputs for agent (process C) and requirements/additions to the knowledge database implied by them.

The following tables are representing correlations between message and the additional knowledge that it brings into internal state machine of process C and the knowledge required to send such messages.

Data values semantics

We will be using data values unused till this moment i.e `ANY` and `NA`.

- `ANY` represents any data that is known to the process C. It can be a new nonce generated by the agent or any randomized value.
- `NA` represents value that is not known to the process C. It's used in situation when required knowledge conditions to send a message are not met. Its usage simplifies the code.

Agent accumulating knowledge

Received message	Additional knowledge
(X, A, C)	X
(X, A, B)	(X, A, B)
(X, C)	X
(Y, C)	Y
(ANY, C)	-
(A, C)	-
(B, C)	-
(C, C)	-
(Y, Y, C)	Y
(ANY, Y, C)	Y
(X, Y, A)	(X, Y, A)
(Y, Y, A)	(Y, Y, A)
(ANY, Y, A)	(ANY, Y, A)
(A, Y, C)	Y
(B, Y, C)	Y
(C, Y, C)	Y
(A, Y, A)	(A, Y, A)
(B, Y, A)	(B, Y, A)
(C, Y, A)	(C, Y, A)
(X, B)	(X, B)
(Y, B)	(Y, B)
(ANY, B)	(ANY, B)
(A, B)	(A, B)
(B, B)	(B, B)
(C, B)	(C, B)

This table is modeled with the following Promela code:

```
:: d_step {
  chExchange ? _, data1, data2, data3; if
    :: (data3 == C)-> learn1(data1); learn1(data2)
    :: else learn3(data1,data2,data3)
  fi;
  data1 = 0; data2 = 0; data3 = 0;
}
:: d_step {
  chConfirm ? _, data1, data2; if
    :: (data2 == C)-> learn1(data1)
    :: else learn2(data1,data2)
  fi;
  data1 = 0; data2 = 0;
}
```

Learn macros are helper macros to switch bit states representing knowledge of facts:

```
#define learn1(data1) if \
        :: (data1 == X)-> know_X = 1 \
        :: (data1 == Y)-> know_Y = 1 \
        :: else skip \
    fi;

#define learn2(data1,data2) if \
        :: (data1 == Y && data2 == B)-> know_YB = 1 \
        :: else skip \
    fi;

#define learn3(data1,data2,data3) if \
        :: (data1 == X && data2 == A && data3 == B) -> know_XAB = 1 \
        :: (data1 == X && data2 == Y && data3 == A) -> know_XYA = 1 \
        :: else skip \
    fi;
```

Agent using its knowledge

Sent message	Required knowledge
(X, A, B)	X or (X, A, B)
(X, B, B)	X or (X, B, B)
(X, C, B)	X or (X, C, B)
(Y, A, B)	Y or (Y, A, B)
(Y, B, B)	Y or (Y, B, B)
(Y, C, B)	Y or (Y, C, B)
(ANY, A, B)	-
(ANY, B, B)	-
(ANY, C, B)	-
(X, A, A)	X or (X, A, A)
(X, B, A)	X or (X, B, A)
(X, C, A)	X or (X, C, A)
(A, A, B)	-
(A, B, B)	-
(A, C, B)	-
(B, A, B)	-
(B, B, B)	-
(B, C, B)	-
(C, A, B)	-
(C, B, B)	-
(C, C, B)	-
(Y, B)	Y or (Y, B)
(X, X, A)	X or (X, X, A)
(X, Y, A)	(X & Y) or (X, Y, A)
(X, ANY, A)	X or (X, ANY, A)

The table is modeled with the following Promela code:

```
/* ... */
    :: chExchange ! ((kX || k_X_A__B) -> B : NA), X, A, B
    :: chExchange ! (kX -> B : NA), X, B, B
    :: chExchange ! (kX -> B : NA), X, C, B
    :: chExchange ! (kY -> B : NA), Y, A, B
```

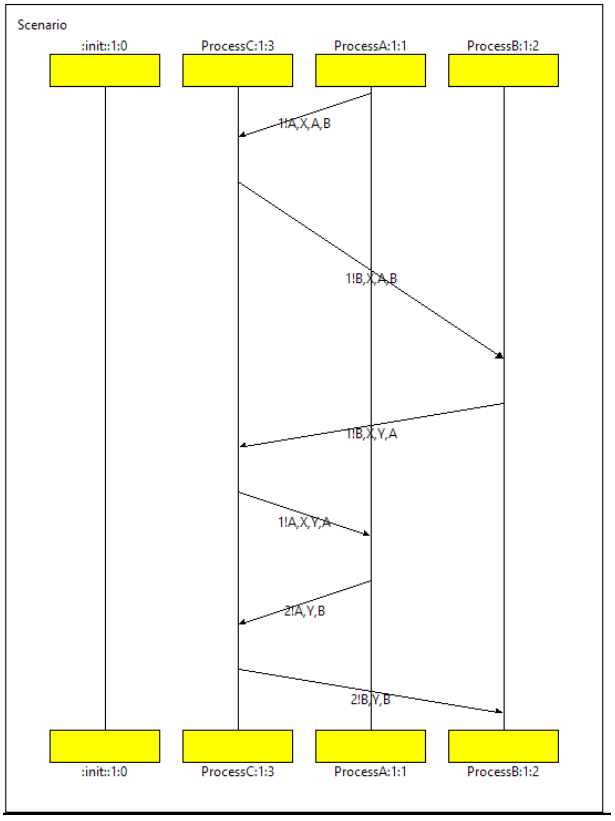
```
    :: chExchange ! (kY -> B : NA), Y, B, B
    :: chExchange ! (kY -> B : NA), Y, C, B
    :: chExchange ! B, ANY, A, B
    :: chExchange ! B, ANY, B, B
    :: chExchange ! B, ANY, C, B
    :: chExchange ! (kX -> A : NA), X, A, A
    :: chExchange ! (kX -> A : NA), X, B, A
    :: chExchange ! (kX -> A : NA), X, C, A
    :: chExchange ! B, A, A, B
    :: chExchange ! B, A, B, B
    :: chExchange ! B, A, C, B
    :: chExchange ! B, B, A, B
    :: chExchange ! B, B, B, B
    :: chExchange ! B, B, C, B
    :: chExchange ! B, C, A, B
    :: chExchange ! B, C, B, B
    :: chExchange ! B, C, C, B
    :: chConfirm ! ((k_Y__B || kY) -> B : NA), Y, B
    :: chConfirm ! ((k_Y__B || kY) -> B : NA), Y, B
    :: chExchange ! (kX -> A : NA), X, X, A
    :: chExchange ! (((kX && kY) || k_X_Y__A) -> A : NA), X, Y, A
    :: chExchange ! (kX -> A : NA), X, ANY, A
/* ... */
```

Finding the solution

To find the solution please run the following command:

```
$ bash ./smart_c_find.sh
```

The Spin tool will generate a trace with the result.



The diagram differs only in layout terms from the top one, which means we found a working MITM trace!

Modelling CA proccess

We want to add A CA process to the model, such `CA` will communicate with all parties, but `C` won't intercept messages on lines `A<->CA` or `B<->CA`.

We will change hardcoded `A` and `B`s on last element of message tuples to values received exactly once during process lifetime - when the public keys are required to continue for the first time.

You can see the example with CA process here: with_ca.pml

Trail for that example can be found here

The diagram is nearly the same, but with queries back and forth to `CA`.

Modelling CA middleman

We can model `MIDM` on `A/B<->CA` connections as well:

```
        /* Channels for incoming and outcoming requests for pubkey */
        chan chCAIn = [0] of {
                mtype:DataType, mtype:CAName
        };
        chan chCAOut = [0] of {
                mtype:DataType, mtype:DataType
        };
```

CA process iself looks like this:

```
/* Process that answers with public keys */
proctype ProcessCA() {
        mtype:CAName chReqIn;
        do
                :: (processASuccess && processBSuccess) -> break;
                :: else ->
                        chCAIn ? CA, chReqIn;
                        if
                                :: (chReqIn == CA_A) -> chCAOut ! CA, A;
                                :: (chReqIn == CA_B) -> chCAOut ! CA, B;
                        fi
        od
}
```

And our agent can send, receive and change messages:

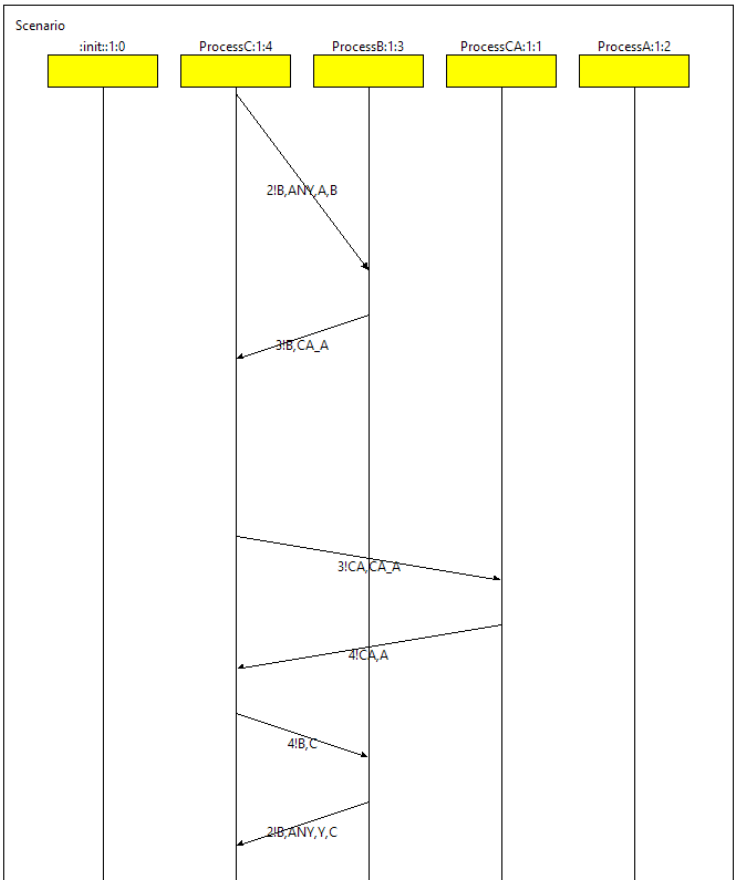
```
        /* ... */
        :: atomic {
                chCAIn ? B, chReqInC;
                chCAIn ! CA, chReqInC;
        }
        :: atomic {
                chCAOut ? CA, chReqInC;
                chCAOut ! B, chReqInC;
        }
        :: atomic {
                chCAOut ? CA, chReqInC;
                chCAOut ! B, C;
        }
        /* ... */
```

As earlier you can generate trace for that model simply calling:

```
$ bash ./broken_ca_find.sh
```

You can find the example in the broken_ca.pml file. Trace is available here - broken_ca.pml.trail with MSC diagram as well - broken_ca.msc

As we may expect the trace diagram looks like follows:



The agent swaps pubkeys and now the knowledge stat is as follows:

```
know_X = 1
know_Y = 1
```

That's great! We just swapped the public keys with our own and the agent managed to completely compromise the connection! Hooray! 🚀

Piotr Styczyński 25-04-2019