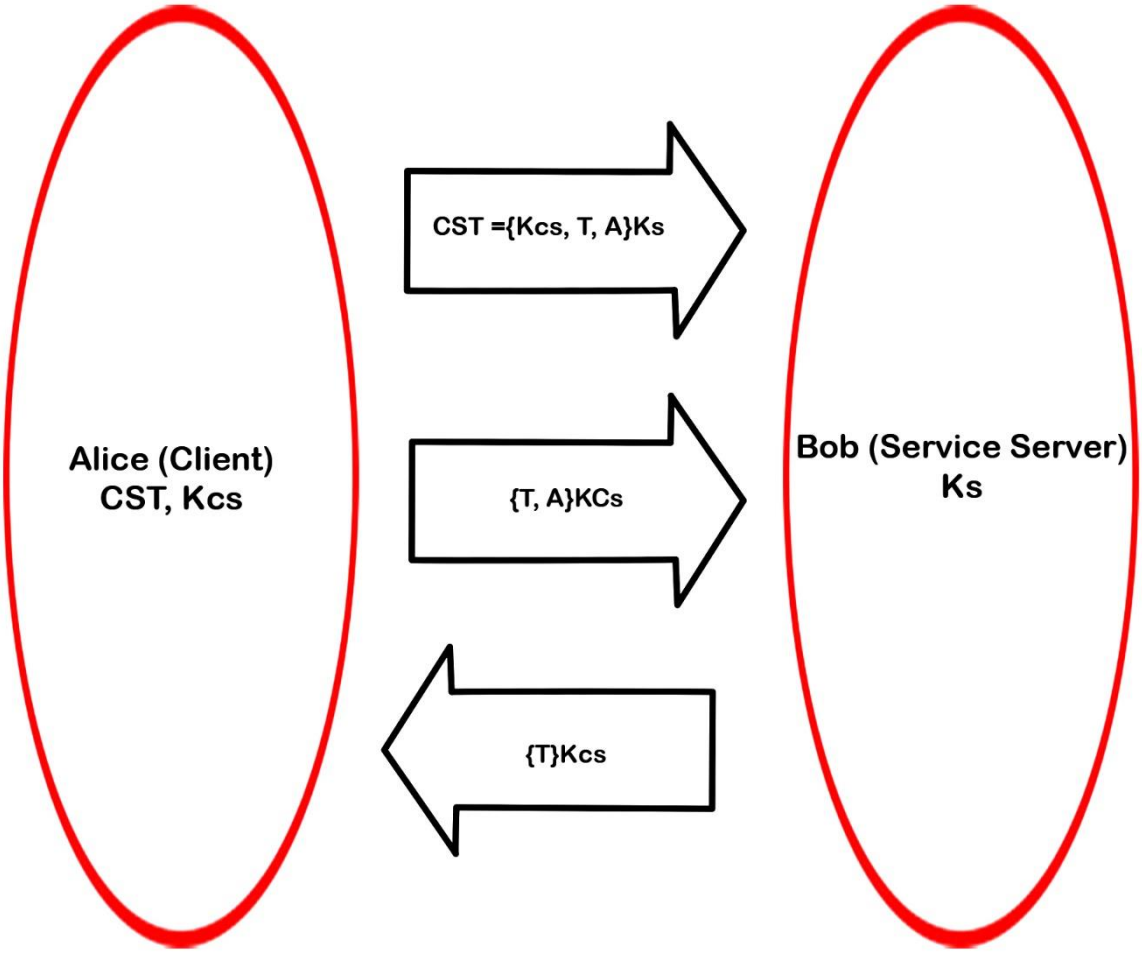
Data Security for Embedded Systems – Lab 2

Mohammad Hammoud & Yaman Tallozi

1. **In Lecture 7 we discussed the Kerberos architecture and protocol. Using the description of the protocol from https://en.wikipedia.org/wiki/Kerberos_(protocol) draw the Client Service Request step diagram following the same pattern and symbols as with the other steps during the lecture (this is in the vicinity of slide 26).**



2. **In the file area for your lab group on Blackboard you can find the lab2.zip file. This file is group specific and you have to use it for solving this assignment. In this file there is a pre-compiled Secret.class file and the Main.java file that shows the sample usage of the interface of the the Secret class. The Secret class has only two public methods**

   The password is: B797C6A76C2C0D3F48DCC1468C7B3283B43F3495

   Check Main.java file

3. **In Lecture 9 we discussed the timing and information leak in the mutual authentication step of the BAC protocol quoting a possible code revealing the weakness (slide ~41 on). Explain how to fix this implementation issue, provide fixed code (does not have to compile or be proper Java (Card), can be pseudo-code) that eliminates both issues.**

   pseudo-code:

   1. Generate random time

```
int rand = Math.rand(0xffff);

for(int i = 0; i < rand; i++);
```

   2. Checkmac

```
boolean msg = false;

if (!checkMAC(apdu, key)) {msg = true; }
```

   3. Checkchellange

```
if (myN != N) {msg = true; }
```

   4. If any of them are wrong, throw exception “Wrong MAC or nounce”

```
If( msg ){

        throw new Exception("Wrong MAC or nounce ");

}
```

   Generating random time hides the time for checking of mac and decryption get hidden.

   The error message is the same of both mac and nounce so attacker will not exactly know if the error from mac or from nounce.

4. **Consider the file PIN.java file attached to these exercises on Blackboard. Provide a set of unit tests (preferably using JUnit) that would test the method checkPin(int pin) according to the MC/DC requirements. Explain how you chose the test data and constructed the corresponding test cases to satisfy the MC/DC testing criteria. Hint: The PIN class interface does not allow you to change the PIN state directly, this is on purpose as it would be insecure. To allow the PIN object to be in different states, you can follow two strategies, choose one: ○ Initialise a default PIN object and bring it to the desired state using the available API (for example, if you need to have it with only one try left, construct a PIN object and provide a wrong PIN twice). ○ Use the Java reflection API to manipulate the internals of the PIN object directly.**

Explanation:

we have three values represented in **access**, **securePin** and **tryCounter**. In order to pass, combination of the three should return a true value, presented in the following:

- **access** returns a true value if two conditions happen, access or tryCounter is bigger than zero AND pin is equal to the securePin
- **securePin** returns true if the choosing pin is equal to the securePin, in this case pin = 1297.
- **tryCounter** returns true if it's a positive value AND bigger than zero.

The tested combinations here are 8 situations: 000, 001, 010, 011, 100, 101, 110, 111

Access	tryCounter	securePin	Return
0	0	0	FALSE
0	0	1	FALSE
0	1	0	FALSE
0	1	1	TRUE
1	0	0	TRUE
1	0	1	TRUE
1	1	0	TRUE
1	1	1	TRUE

MC/DC

(access): 1 and 5

2 and 6

3 and 7

(tryCounter): 2 and 4

(securePin): 3 and 4

Situation 5 and 6 cannot be tested, because the access can't be true at the same time when **tryCounter** is negative.

5. **In the later parts of the course we discussed how to use ProVerif to verify the security of the handshake protocol between Alice and Bob. In the process we eliminated some flaws and proved some security properties, however, we ended up with a protocol that still has the replay attack on Alice possible. The complete ProVerif file is attached to the exercise sheet. Fix the protocol to prevent replay attack on Alice (hints on how to are included in the lecture), check it with ProVerif (you can use the on-line interface at http://proverif16.paris.inria.fr/) that it is so**

So first generate a new nonce at the beginning of her process

Alice sends this nonce to Bob

Bob receives the nonce and puts this in the signature with everything else

Alice checks that the nonce that she receives from Bob is the same as she generated

Fixed Code:

```
type key.

fun senc(bitstring, key): bitstring.

reduc forall m: bitstring, k: key; sdec(senc(m, k), k) = m.

type skey.
type pkey.
type nonce.

fun pk(skey): pkey.
fun aenc(bitstring, pkey): bitstring.

reduc forall m: bitstring, k: skey; adec(aenc(m, pk(k)), k) = m.

type sskey.
type spkey.

fun spk(sskey): spkey.
fun sign(bitstring, sskey): bitstring.

reduc forall m: bitstring, k: sskey; getmess(sign(m, k)) = m.
reduc forall m: bitstring, k: sskey; checksign(sign(m, k), spk(k)) = m.

free c: channel.
free s: bitstring [ private ].

event aliceReceiveKey(key).
event bobFreshKey(key).

query attacker(s).

query x: key; inj-event(aliceReceiveKey(x)) ==> inj-event(bobFreshKey(x)).


let alice(pkA: pkey, skA: skey, pkB: spkey) =
  new nc: nonce;
  out(c, nc);
  out(c, pkA);
  in(c, x: bitstring);
  let sigB = adec(x, skA) in
  let (=pkA, =pkB, =nc, k: key) = checksign(sigB, pkB) in

  event aliceReceiveKey(k);
```

```
  out(c, senc(s, k)).

let bob(pkB: spkey, skB: sskey) =
  in(c, nn: nonce);
  in(c, pkX: pkey);
  new k: key;
  event bobFreshKey(k);
  out(c, aenc(sign((pkX, pkB, nn, k), skB), pkX));
  in(c, x:bitstring);
  let z = sdec(x, k) in
    0.

process
  new skA: skey; new skB: sskey;
  let pkA = pk(skA) in out(c, pkA);
  let pkB = spk(skB) in out(c, pkB);
  ( (!alice(pkA, skA, pkB)) | (!bob(pkB, skB)) )
```

ProVerif text output:

ProVerif text output:

```
-- Process 1-- Query not attacker(s[]) in process 1
Translating the process into Horn clauses...
Completing...
Starting query not attacker(s[])
RESULT not attacker(s[]) is true.
-- Query inj-event(aliceReceiveKey(x_2)) ==> inj-event(bobFreshKey(x_2)) in process 1
Translating the process into Horn clauses...
Completing...
Starting query inj-event(aliceReceiveKey(x_2)) ==> inj-event(bobFreshKey(x_2))
goal reachable: begin(bobFreshKey(k_2),@occ22_1) -> end(@occ15_1,aliceReceiveKey(k_2))
The hypothesis occurs strictly before the conclusion.
Abbreviations:
nc_1 = nc[!1 = @sid]
k_2 = k_1[pkX = pk(skA[]),nn = nc_1,!1 = @sid_1]
@occ15_1 = @occ15[x = aenc(sign((pk(skA[]),spk(skB[]),nc_1,k_2),skB[]),pk(skA[])),!1 = @sid]
@occ22_1 = @occ22[pkX = pk(skA[]),nn = nc_1,!1 = @sid_1]
RESULT inj-event(aliceReceiveKey(x_2)) ==> inj-event(bobFreshKey(x_2)) is true.

------------------------------------------------------------
Verification summary:

Query not attacker(s[]) is true.

Query inj-event(aliceReceiveKey(x_2)) ==> inj-event(bobFreshKey(x_2)) is true.

------------------------------------------------------------
```