Network Security Class
Lab Session 3

Stefano Zanella - 621796

July 2013

Network Security class

Nicola Laurenti, 2012-13

Laboratory session 3

student: _____ ID: _____

A and B want to establish a secure connection based on symmetric cryptography, sharing a secret key k. In order to do that, since each of them has a connection with asymmetric cryptography to C, they want to use the following protocol:

k_{A}	private key of A
k'_{A}	public key of A (known to C)
k_{B}	private key of B
k'_{B}	public key of B (known to C)
k_{C}	private key of C
k'_{C}	public key of C (known to A and B)

1. A : generates nonce $r_{\mathsf{A}} \sim \mathcal{U}(\mathcal{R})$
   $\mathsf{A} \to \mathsf{C}$: $[\mathrm{id}_{\mathsf{A}}, \mathrm{id}_{\mathsf{B}}, r_{\mathsf{A}}]$

2. C : generates $k \sim \mathcal{U}(\mathcal{K})$, $u_1 = [k, r_{\mathsf{A}}]$, encrypts $x_1 = E_{k'_{\mathsf{A}}}(u_1)$
   $\mathsf{C} \to \mathsf{A}$: x_1
   A : decrypts $u_1 = D_{k_{\mathsf{A}}}(x_1)$

3. A : signs $t_2 = S_k([\mathrm{id}_{\mathsf{A}}, \mathrm{id}_{\mathsf{B}}, r_{\mathsf{A}}])$
   $\mathsf{A} \to \mathsf{B}$: $x_2 = [\mathrm{id}_{\mathsf{A}}, \mathrm{id}_{\mathsf{B}}, r_{\mathsf{A}}, t_2]$

4. $\mathsf{B} \to \mathsf{C}$: $[\mathrm{id}_{\mathsf{A}}, \mathrm{id}_{\mathsf{B}}, r_{\mathsf{A}}]$

5. C : encrypts $x_3 = E_{k'_{\mathsf{B}}}(u_1)$
   $\mathsf{C} \to \mathsf{B}$: x_3
   B : decrypts $u_1 = D_{k_{\mathsf{B}}}(x_3)$

6. B : verifies $b = V_k([\mathrm{id}_{\mathsf{A}}, \mathrm{id}_{\mathsf{B}}, r_{\mathsf{A}}], t_2)$

Your assignment is to:

1) implement the above protocol in your favourite language, and check its correctness;
2) identifiy its vulnerabilities and devise an attack that exploits them, under reasonable assumptions;
3) implement the attack and evaluate its success probability in dependence of the protocol parameters;
4) suggest improvements to the protocol and implement them.

Provide a description of your solution, with justification of your choices, the code for your implementations, and adequate discussion of the results.

General notes: Implementing your protocol may require you to use several cryptographic primitives, such as symmetric or asymmetric encryption/decryption, signing/verification, one-way functions, cryptographic hash functions, key generation, etc. Feel free to use any reasonable (and correct) implementation of such primitives you find for the language you've chosen, or even simplified models such as a (pseudo-)random oracle. The vulnerability of the protocols should not depend on a poor implementation of such primitives. Also, unlike for cryptographic keys, you should model the random choice of a password as (strongly) non uniform.

1 Implementation Details

Here are the choices made where requested by the assignment or when no further specification given:

- The assignment has been implemented in **Ruby** (version **2.0.0-p195**)
- To simplify the resulting codebase hosts have been modeled as classes, and communications over the network channel have been modeled with a `Channel` class that acts like a simple shared message storage. This allows to easily simulate message exchanging and eavesdropping.
- For the same reason, and to make the source code more readable, data concatenation has been modeled with hash maps. This in fact replaces specifying data chuncks' lengths in the protocol with specifying the keys at which data chuncks are accessible.
- Cryptographic primitives (ciphers, RNGs, etc.) are provided by Ruby's wrapper around the **OpenSSL** library and core language facilites. In detail:
  - RNG is provided by the built-in `Random` class (documentation), a PRNG based on a *Mersenne twister*.
  - asymmetric public key algorithm is RSA (Ruby documentation), with **2048 bits** key length.
  - symmetric key cipher is **AES 256**, used when simulating communications between nodes.
  - message authentication/integrity protection is provided by class `OpenSSL::HMAC`, which accepts an available digest algorithm as a parameter. Selected cryptographic hash function for message authentication is **SHA512**, provided by class `OpenSSL::Digest::SHA512` (documentation).
- Since OpenSSL RSA works on strings, all hashes used for communication needed to be serialized before encryption. Ruby's built-in standard for serialization/deserialization is **YAML** (documentation), which use is made transparent by encapsulation into class `NetSec::Node`.

2 Protocol Implementation

Given protocol is implemented in `NetSec::KeyExchange#start!` method. Correctness is proven at the end by printing keys hold by **A** and **B**, which are obviously the same in case the protocol works correctly.

To run the exchange, just type at the prompt, from inside the source folder:

```
bin/key_agreement
```

The output should be something similar to:

```
A has key: ["a2b85f3a7164c411d8ea5eb66affa741472eb59be787
                                    c3b3ea63b7816c868d39"]
B has key: ["a2b85f3a7164c411d8ea5eb66affa741472eb59be787
                                    c3b3ea63b7816c868d39"]
```

3 Possible Flaws and Related Attacks

C Node Spoofing

While **C** makes use of asymmetric cryptography to exchange the key with **A** and **B**, these last two does not the same when communicating with **C**. This easily allows an attacker to spoof **C**'s identity (for example, by DoSing it and routing requests to itself); once the attacker can successfully impersonate **C** it has just to save the list of generated keys and start eavesdropping from the communication channel. This way it can do whatever it wants (from simply logging exchanged information to manipulation of exchanged data), given no other security mechanisms are in place for a given session (e.g. for message integrity).

4 Attacks Implementation and Analysis

C Node Spoofing

To simulate spoofing of node C, a `NetSec::SpoofedC` subclass has been introduced. Basically it acts the same way as its parent class, plus it eavesdrop on

the channel upon initialization and then saves the generated key to decrypt eavesdropped messages.

Spoofing simulation can be run from the prompt by invoking:

```
bin/spoofing_attack
```

which outputs something along the lines of:

```
A has key: ["c26c3f32631b6eb395d90a63f835baccbdbc244cd400
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣a15312bbfea15bffa93b"]
B has key: ["c26c3f32631b6eb395d90a63f835baccbdbc244cd400
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣a15312bbfea15bffa93b"]
Spoofed C has key: ["c26c3f32631b6eb395d90a63f835baccbdbc
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣244cd400a15312bbfea15bffa93b"]
B received: My credit card number is 1234567890123456
The attacker eavesdropped: My credit card number is
                                                1234567890123456
```

5 Possible Improvements

C Node Spoofing

A simple solution to the problem of spoofing **C** identity would be to make use of **C** asymmetric keys during key agreement.
In particular, in steps **1** and **4**, **A** and **B** could encrypt the message $[id_A, id_B, r_A]$ using **C**'s public key k'_C. On the other side, **C** would then decrypt received messages using its private key k_C. With this simple improvement, the only way for an attacker to perform the same spoofing attack would be to steal **C**'s private key, which is supposed to be an event with low probability of success given the assumptions on which asymmetric cryptography is based.

This solution is implemented in classes `AntispoofingA`, `AntispoofingB` and `AntispoofingC`. The correctness of the implementation can be seen by launching from the prompt:

```
bin/antispoofing_agreement
```

To simulate the attack against this improved version, class `SpoofedAntispoofingC` has been created. By launching

```
bin/antispoofing_attack
```

it can be seen how the whole process generates an error when the malicious **C** tries to decrypt the message from **A**'s step 1 without having the correct private key.