# 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:

| | |
|---|---|
| $k_{\mathsf{A}}$ | private key of A |
| $k'_{\mathsf{A}}$ | public key of A (known to C) |
| $k_{\mathsf{B}}$ | private key of B |
| $k'_{\mathsf{B}}$ | public key of B (known to C) |
| $k_{\mathsf{C}}$ | private key of C |
| $k'_{\mathsf{C}}$ | public key of C (known to A and B) |

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

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

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

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

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

$\boxed{6}$ 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 protocol 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 Protocol Flaws and Possible Attacks

Before describing flaws and possible attacks against them, let's just point out a detail that is not made explicit in the assignment but seems reasonable to assume since otherwise the protocol would be useless.

At **step 2**, after decrypting $x_1$ we'll assume that A checks that the received $r_A$ is equal to the one generated at step 1. If this check is not done, then there's no point in generating the nonce: at that point, even with the improvements depicted below, it will still be possible for an attacker to impersonate node C and send an arbitrary key to A. The same reasoning applies symmetrically to B at **step 5**.

Also, we'll assume that at every step which involves sending identifiers (such as step 1), the receiver verifies the id of the sender; this makes spoofing harder and provides another level of security, easily achievable.

## 3.1 C Node Spoofing

### Attack description

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).

### Attack implementation and analysis

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
```

### Possible improvements

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.

## 3.2 Forging Attack on A

### Attack description

The following is not properly an attack, rather an omissis in the assignment that has an obvious answer. However, let's see what could happen anyway.

Suppose that verification of the nonce by A at **step 2** fails. Also suppose A wants to retry the agreement; the implementation needs to answer a question: would A retry to send the packet and see if C answers correctly or would she restart the agreement with a new nonce?
The answer is the latter, because if A is allowed to retry sending indefinitely the same message $[id_A, r_A]$, then an attacker impersonating C, even if not possessing $k'_C$ (but possessing $k'_A$ which we anyway assume to be *public*), can try to forge $[k, r_A]$ until it's accepted by A. In the context of the simulation failures and retries are not considered to keep complexity low, but this issue can easily be addressed once we're able to identify the failure (more on this later).

## 3.3 KPA Attack

### Attack description

In given implementation, there is room for a KPA attack on $k'_C$: if we consider C node spoofing solved (see above), then an attacker can gain pairs of known plain and ciphertext by gathering the message sent at step 1 from A to C, and the signed message sent from A to B at step 3. By discarding the signature, the attacker has access to a potentially unlimited amount of pairs (since keypairs aren't changed often) that can be used for an offline attack against C's private key.

### Attack implementation and analysis

The actual implementation and success probability of such an attack isn't related to the nature of the protocol, but rather depends on the specific asymmetric cryptohgraphy scheme used in protocol implementation. Since it's assuming (per assignment suggestion) that protocol vulnerability doesn't depend on poor implementation of cryptographic primitives, a complete implementation of such an attack is not provided. Though, it's interesting to note that this attack is much more practical than one might think. Without straying from proposed implementation where RSA is used, for example, it's proved in [1] that a successful attack can be carried out even by knowing just a partial plaintext (although in that paper there's no mention about practical implementations of such an attack and supposed complexity with respect to key length).

### Possible improvements

A solution of this issue, which also brings half of another desirable feature, is to:

- modify step 2 so that C also sends $k'_B$ to A
- modify step 3 so that A sends $\hat{x_2} = E_{k'_B}(x_2)$ instead of $x_2$
- modify step 4 so that B first decrypts $x_2 = D_{k_B}(\hat{x_2})$, then sends $x_2$ to C

The advantage of this approach is that it allows A to authenticate B, since only B is supposed to be able to decrypt $\hat{x_2}$.

This solution is implemented in classes `AntispoofingNoKPAA`, `AntispoofingNoKPAB` and `AntispoofingNoKPAC`. The correctness of the implementation can be seen by launching from the prompt:

```
bin/antispoofing_nokpa_agreement
```

# 4 Furhter improvements

We've stated that, in the attempt to avoid the possibility of a KPA, we've introduced authentication of B to A. While not strictly needed, we can think of also performing the other side of the authentication by symmetrically adapting the protocol on the B side. That is, we could let C send to B $k'_A$ at step 5; then, after correctly verifying signature from A, B could simply send a challenge $E_{k'_A}(r_B),\ r_B \sim \mathcal{U}(\mathcal{R})$. At this point, A would just decrypt $r_B$ and send back $E_k(r_B - 1)$ (so to check that $k$ works as expected). This way the protocol would also implement mutual public key authentication (again, not strictly needed given the purpose of the protocol).

# 5 Open Question

An open question remains, which doesn't count too much for the purposes of this simulation, but gains relevance as soon as we try to adapt protocol to real world, which is: how to handle and notify failures in key agreement in a secure way?

That is, in the simulation when a failure happens due to a failed attack, it's easy to stop the agreement by just raising an exception. This works well because we're running inside a single program, but can't be a solution when nodes are physically distinct machines. As long as the failure happens at A the problem doesn't arise: since it's A to have initiated the agreement, it can just retry as explained or stop trying.
But, suppose a failure happens at C in the middle of the exchange with A: what should C do? Probably the safest thing to do is to just not send any answer, letting A wait and eventually decide to drop the tentative by timeout. But, if A is a legitimate sender, then such behaviour would result in a loss of performance, which may or may not be mitigated/accepted with timeout tuning. On the other side, letting A know explicitly about the failure, would give information to an eventual attacker that could modify his success probability to his advantage.
The same question applies when the failure happens at B, which also arise the problem of how to let A know of the failure, again without giving a possible attacker any specific information that could be used to his advantage.

## 6 References

1. *Sahadeo Padhye*, Partial Known Plaintext Attack on Koyama Scheme. School of Studies in Mathematics, Pt.Ravishankar Shukla University, Raipur (C.G.), India, 2002 (online version).