

1-02-328: Communication and E-Commerce Security

Recitations 9–10: Signing Protocols

Michael J. May

May 4, 2011

In this recitation we will write some code which uses the .NET cryptographic libraries for signatures and signature verification. We will then use these tools to generate mockups of some of the protocols we have seen and discussed in class.

1 Introduction

In class we talked about two hash functions, SHA1 and MD5. We didn't go into the details of how they are computed, but both functions take a variable length input and produce a fixed length output. Microsoft .NET offers managed implementations of both MD5 and SHA1 for you to use to compute hashes. Hash verification is fairly simple - just examine whether the computed hash value is equal to the value it is supposed to have.

We discussed digital signatures last lecture as part of a discussion about how we can verify data being sent over an insecure network. We talked about symmetric and asymmetric digital signatures, both of which amount to an encryption of a summary of a message. Microsoft .NET includes a digital signature implementation that does all of the work for us - hashing and signing at once.

2 Public Key Signatures

In order to perform public key signatures and signature verification we must ensure that both sides of the communication protocol (sender and receiver) have a copy of the same public key. That means that in order for Bart to verify Alice's signature he must first have a copy of Alice's **public key**. Similarly, for Alice to verify Bart's signature she needs a copy of Bart's **public key**.

To make things simpler today I have prepared a small program which generates RSA public and private key pairs and saves them as XML files. The tool can save just the public part of the RSA key pair or both the public and private parts. A screen shot of the tool is shown in Figure 1.

The code for the tool is available on the course web site as well if you are interested in seeing how it works.

You should use the tool to generate two key pairs, one for Alice and one for Bart.

Alice's Keys For Alice, generate a 512 bit RSA key pair. Save her key pair twice:

- (1) Save it once **with** the private key information. Call the file `AlicePrivate.xml`.
- (2) Save it once **without** the private key information. Call the file `AlicePublic.xml`.

Bart's Keys For Bart, generate a 1024 bit RSA key pair. Save her key pair twice:

- (1) Save it once **with** the private key information. Call the file `BartPrivate.xml`.
- (2) Save it once **without** the private key information. Call the file `BartPublic.xml`.

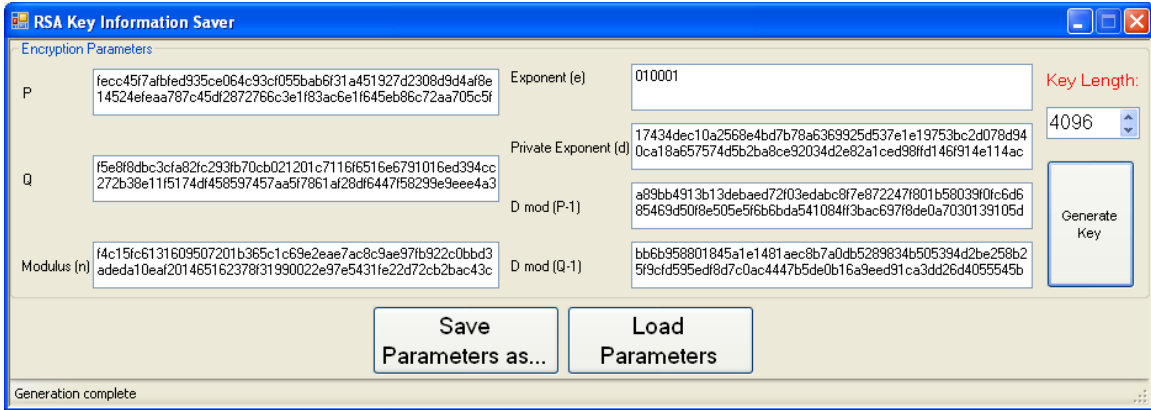


Figure 1: RSA Key Pair Saving and Loading Tool

3 Protocol Version 1

We will start with the most basic form of the public key signature protocol for authentication between Alice and Bart. Alice sends a signed message to Bart saying “Hello from Alice”, Bart checks the signature, and then returns a message to Alice saying “Hello from Bart”. Alice checks Bart’s signature and thereby verifies that the message came from him. A graphical illustration of the protocol is shown in Figure 2. To make the communication a bit simpler, the message is sent in two separate strings via StreamReader/StreamWriter. The first string is the message and the second is a base64 character string encoding of the `byte[]` array of the digital signature.

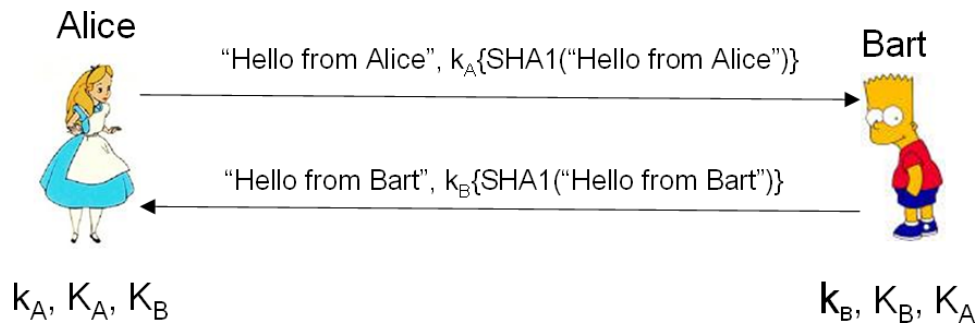


Figure 2: Version 1 of the signing protocol

The steps we must take for the protocol are as follows:

1. Alice must load her public and private keys from the `AlicePrivate.xml` file. She must load Bart’s public key from the `BartPublic.xml` file.
 - The loading is done automatically from the `App.config` file using customizable names and directories. The function that does the work is called `bLoadKeys_Click`.
 - Alice’s keys are stored in the object `myKey`.
 - Bob’s keys are stored in the object `serverKey`.
2. Bart must load his public and private keys from the `BartPrivate.xml` file. He must load the other client’s public keys as well. To make it simpler, the Bart server uses an `App.config` file to direct the Bart application to the directory where the key files are located and a `ClientList` file which tells the

Bart application what its default name is and where its own keys can be found. The loading is done automatically by clicking on the button *Reload Keys* on the application. The work is done in the event handler `bReload_Click`.

- Each client key is loaded using the client name given in the client list file and is stored in a `HashTable`.
 - The key is looked up in the `HashTable` based on the name supplied by the client.
3. Alice must prepare the message “msg1” to send to Bart.
 - The message is prepared in the function `GetMsg1` in the file `AliceClient.cs`.
 4. Alice must **sign** “msg1” with her private key (`aliceKey`).
 - The function for the signature takes a `byte []`, so we use the `UnicodeEncoding.Unicode.GetBytes()` function to extract a `byte` array from “msg1”.
 - The function for signing is called `aliceKey.SignData`. It also returns a `byte` array.
 5. Alice must send the message “msg1” and the signature on the message to Bart.
 6. When Bart receives the message, he must **verify** the signature that he received on “msg1”
 - The client protocol management is done in the function `HandleClient`.
 - The client name (Alice) is looked up in the `HashTable` and the appropriate key extracted.
 - The signature is received, decoded from base64 character encoding, and then checked using the `VerifyData` function.
 7. Bart then prepares “msg2” to send to the client (Alice).
 8. Bart must **sign** “msg2” with his private key (`myKey`).
 - The function for signing is called `myKey.SignData` and takes a `byte []` so we must use `UnicodeEncoding.Unicode.GetBytes` to get a `byte` array from “msg2”
 9. Bart must send the message “msg2” and the signature to Alice.
 10. When Alice receives the message, she must **verify** the signature that she received on “msg2”
 - The verification is done in the function `VerifyMsg2` in the file `AliceClient.cs`
 - Alice does this using Barts’s public key `bartKey` and the function `bartKey.VerifyData`.
 - The verification function also uses a `byte []` as input, so we must use the `UnicodeEncoding.Unicode.GetBytes` function to get a `byte` array equivalent for “msg2”.

When both sides have finished the steps above, the protocol is finished.

4 A GUI implementation of the protocol

I have prepared a GUI implementation of the above protocol so that you may try it out and see how the steps are performed. A screenshot of the implementation is shown in Figures 3 and 4.

5 Protocol Modifications

What are the weaknesses of the protocol as designed so far? Is it secure against the Dolev-Yao model of attack? What kinds of attacks are possible?

We will now modify the message generation functions `GetMsg1` and `GetMsg2` as well as the message verification functions `VerifyMsg1` and `VerifyMsg2` to make the protocol more secure.

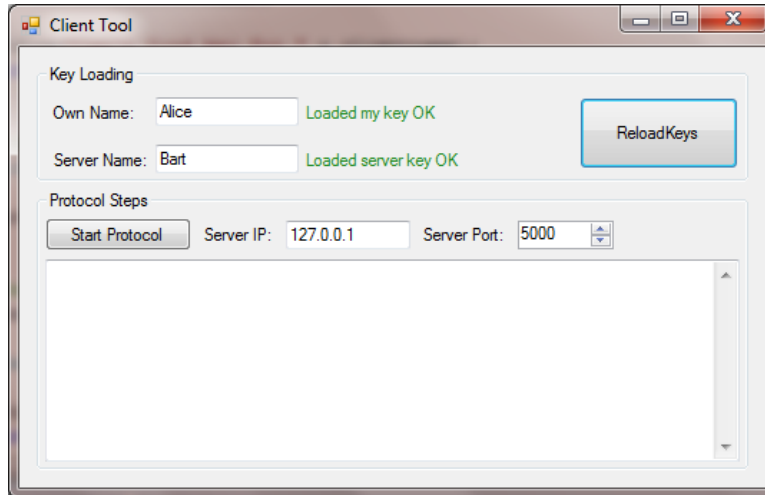


Figure 3: Alice Client Screen Shot

5.1 Protocol Version 2

Modify the generation and verification functions as shown in the slides for version 2, adding the name of the recipient to the message. An illustration of the new protocol version is shown in Figure 5.

1. What attacks have now been prevented?
2. What kind of attacks are still possible?

5.2 Protocol Version 3

Modify the generation and verification functions as shown in the slides for version 3, adding nonces to the protocol (and leaving the name of the recipient as in version 2). An illustration of the new protocol version is shown in Figure 6.

1. What attacks have now been prevented?
2. What kind of attacks are still possible?
3. Why is it necessary to include the third message?

5.3 Protocol Version 4

Modify the generation and verification functions as shown in the slides for version 4, adding a timestamp to the protocol instead of a nonce (and leaving the name of the recipient as in version 2). Make sure the timestamp is accurate to within 1 minute of the time on the recipient's clock. An illustration of the new protocol version is shown in Figure 7.

1. What attacks have now been prevented?
2. What kind of attacks are still possible?
3. Is there a way to avoid the clock synchronization problem?

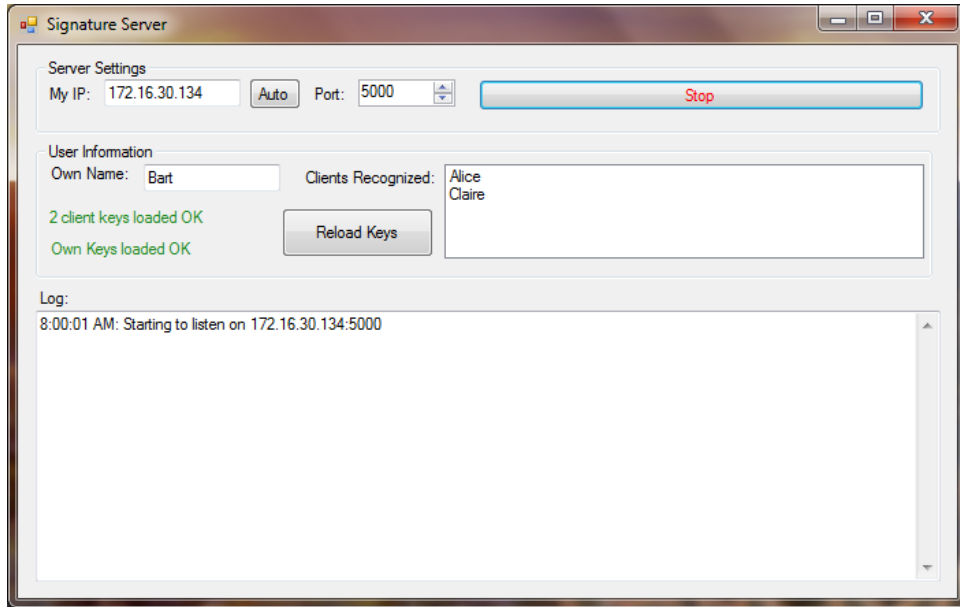


Figure 4: Bart Server Screen Shot

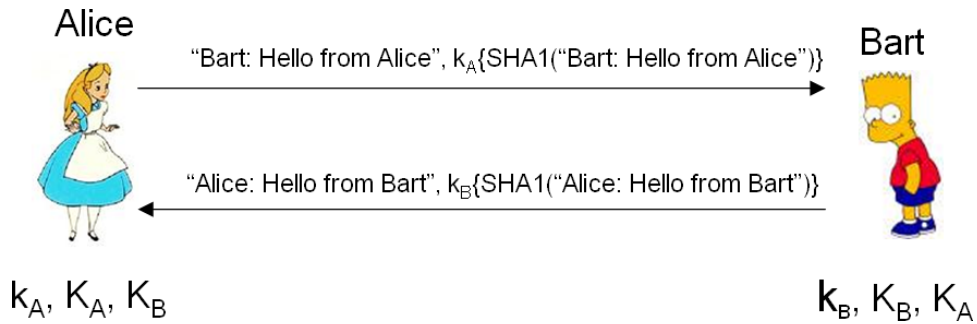


Figure 5: Version 2 of the signing protocol

6 What to do

Your job is to examine the .NET code given for the Protocol Version 1 above and perform the above modifications to make the protocol more secure. With the exception of Version 3, you shouldn't need to touch the code that is in the `bStart_Click` functions, just the `GetMsg1`, `VerifyMsg1`, `GetMsg2`, and `VerifyMsg2` functions. For Version 3, copy the send/receive code found in `AliceClient` and `BartServer` to add another message sent from Alice to Bart.

