

“Course 1-02-328: Communication Security and E-Commerce”

Recitations 9–10: Signing Protocols

Instructor: Michael J. May

May 25 – June 1, 2010

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 A note about about signing files

As discussed in class, most digital signatures are based on a cryptographic hash of the data to be sent. That means that the data cannot be changed while in transit and that the encoding used for interpreting string data must be known. A more interesting problem with signing files is that even seemingly non-changes to a document can cause a digital signature on it to fail. Try the following experiments to get a feeling for how sensitive hashing files is:

2.1 Experiment 1: Text Files

1. Create a new text file called “hash1.txt”. Use notepad to enter the text “This is a file to hash.”
2. Save the file and close notepad.
3. Use the hashing tool we developed in Lab 6 (it's on H: if you can't find yours) to compute the SHA1 hash of the file.
4. After computing the SHA1 hash, reopen “hash1.txt” in notepad and add a space at the end of the file so that it now reads “This is a file to hash. ”
5. Save the file and close notepad.
6. Go back to the Lab 6 hashing tool and click on “Verify Hash”. What is the result?

7. Reopen the file “hash1.txt” in notepad. Remove the trailing space, so it now reads “This is a file to hash.”.
8. Save the file and close notepad.
9. Go back to the Lab 6 hashing tool and click on “Verify Hash”. What is the result?

What have you learned about the storage of text files in Windows?

2.2 Experiment 2: MS Word Files

1. Create a new MS Word file called “hash1.doc”. Use MS Word to enter the text “This is a file to hash.”
2. Save the file and close MS Word.
3. Use the hashing tool we developed in Lab 6 (it’s on H: if you can’t find yours) to compute the SHA1 hash of the file.
4. After computing the SHA1 hash, reopen “hash1.doc” in MS Word and add a space at the end of the file so that it now reads “This is a file to hash. ”
5. Save the file and close MS Word.
6. Go back to the Lab 6 hashing tool and click on “Verify Hash”. What is the result?
7. Reopen the file “hash1.doc” in MS Word. Remove the trailing space, so it now reads “This is a file to hash.”.
8. Save the file and close MS Word.
9. Go back to the Lab 6 hashing tool and click on “Verify Hash”. What is the result?
10. Click on “Compute Hash” again to recompute the hash on the file “hash1.doc”.
11. Reopen the file “hash1.doc” in MS Word. Make no changes to the document. Save the document and close MS Word.
12. Go back to the Lab 6 hashing tool and click on “Verify Hash”. What is the result?
13. Reopen the file “hash1.doc” in MS Word. Add a space to the end of the file so that it reads “This is a file to hash. ”. Then delete the trailing space so that it once again reads “This is a file to hash.”
14. Save the file and close MS Word.
15. Go back to the Lab 6 hashing tool and click on “Verify Hash”. What is the result?

What have you learned about MS Word files?

2.3 Experiment 3: MS Powerpoint Files

1. Download the file “hash1.ppt” from the course web page (it’s inside the ZIP file with the applications). Use the Lab 6 hashing tool to calculate the SHA1 hash of the file.
2. Open the file “hash1.ppt” in MS Powerpoint. Start the slideshow. End the slideshow.
3. Save the file and close MS Powerpoint.
4. Go back to the Lab 6 hashing tool and click on “Verify Hash”. What is the result?

5. Click on “Compute Hash” again to recompute the hash on the file “hash1.ppt”.
6. Reopen the file “hash1.ppt” in MS Powerpoint. Make no changes to the document. Save the document and close MS Powerpoint.
7. Go back to the Lab 6 hashing tool and click on “Verify Hash”. What is the result?

What have you learned about MS Powerpoint files?

3 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.

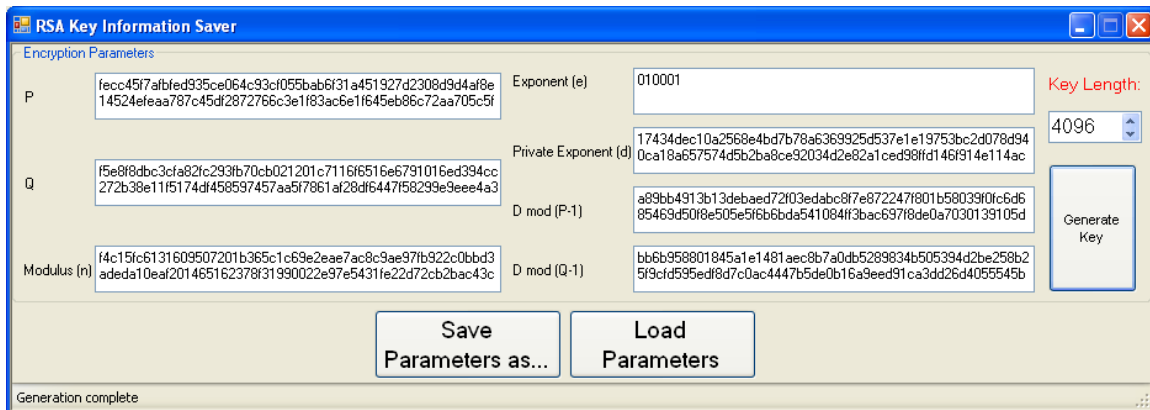


Figure 1: RSA Key Pair Saving and Loading Tool

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`.

4 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.

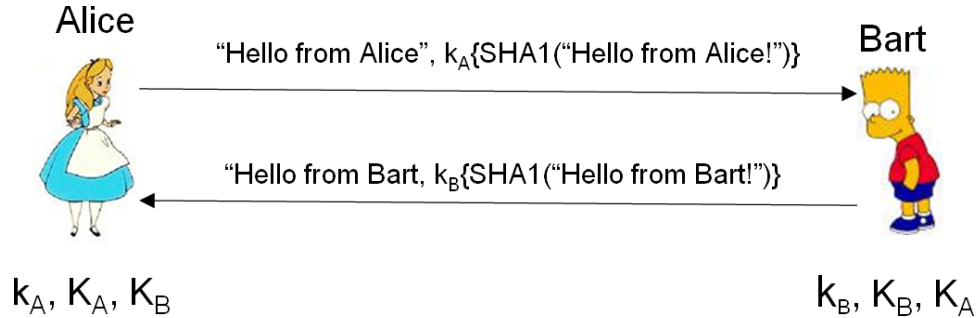


Figure 2: Version 1 of the signing protocol

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.
 - Alice’s keys are loaded in the `bAliceLoad_Click` function in the file `AliceClient.cs`. Her key information is stored in the object `aliceKey`.
 - Bob’s keys are loaded in the `bBartLoad_Click` function in the file `AliceClient.cs`. His key information is stored in the object `bartKey`.
2. Bart must load his public and private keys from the `BartPrivate.xml` file. He must load Alice’s public key from the `AlicePublic.xml` file.
 - Alice’s keys are loaded in the `bAliceLoad_Click` function in the file `BartServer.cs`. Her key information is stored in the object `aliceKey`.
 - Bob’s keys are loaded in the `bBartLoad_Click` function in the file `BartServer.cs`. His key information is stored in the object `bartKey`.
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 verification is done in the function `VerifyMsg1` in the file `BartServer.cs`
 - Bart does this using Alice’s public key `aliceKey` and the function `aliceKey.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 “msg1”.
7. Bart then prepares “msg2” to send to Bart.
 - The message is prepared in the function `GetMsg2` in the file `BartServer.cs`
 8. Bart must **sign** “msg2” with his private key (`bartKey`).
 - The function for signing is called `bartKey.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.

5 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.

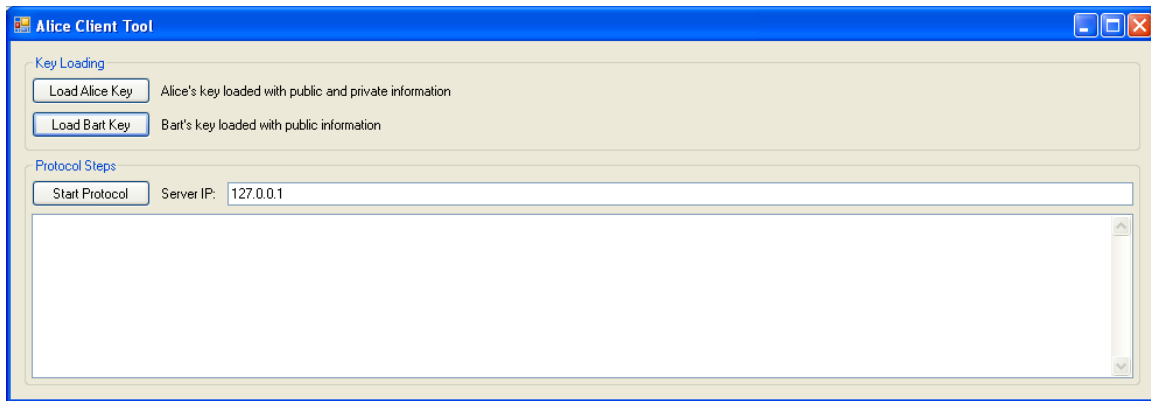


Figure 3: Alice Client Screen Shot

6 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.

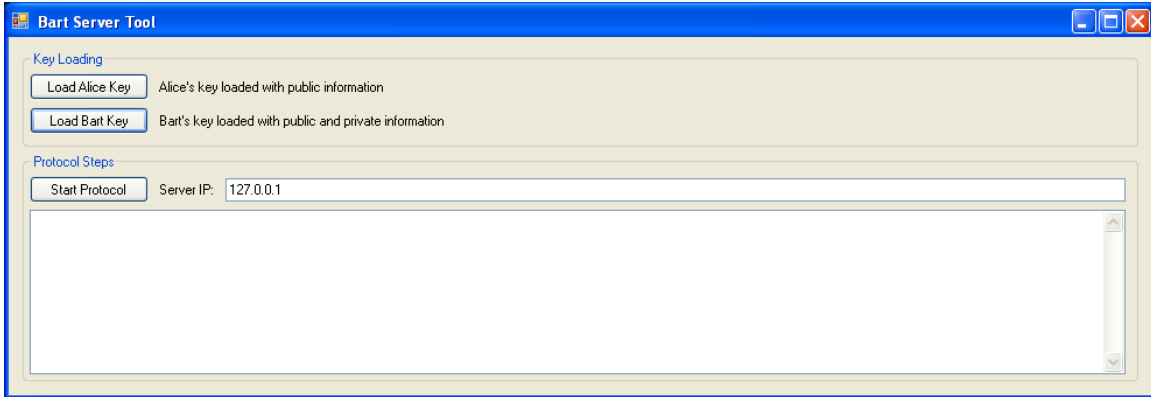


Figure 4: Bart Server Screen Shot

6.1 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.

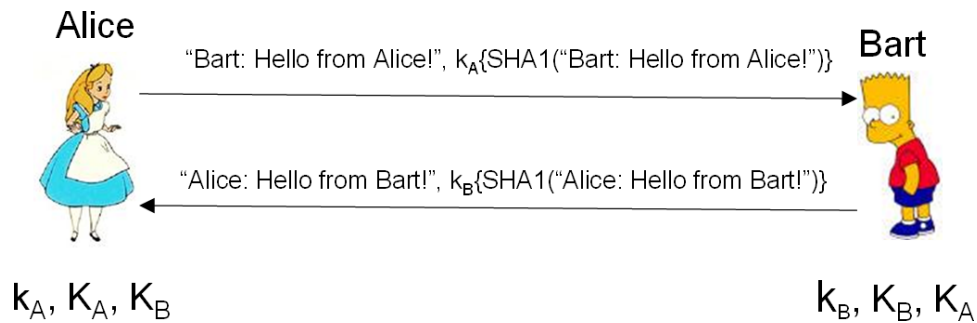


Figure 5: Version 2 of the signing protocol

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

6.2 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?

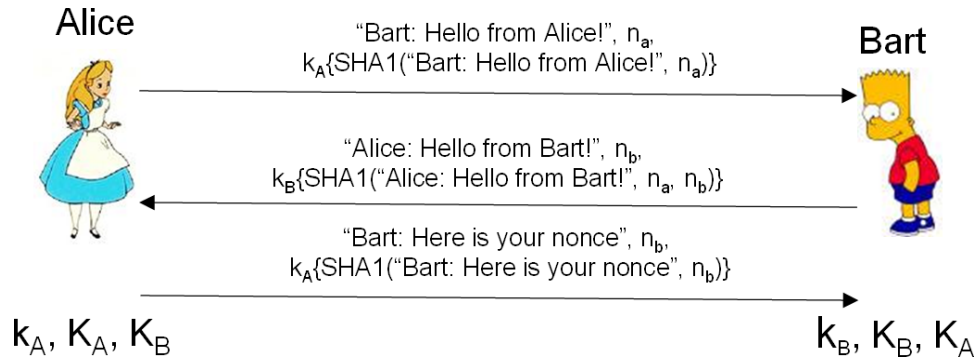


Figure 6: Version 3 of the signing protocol

6.3 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.

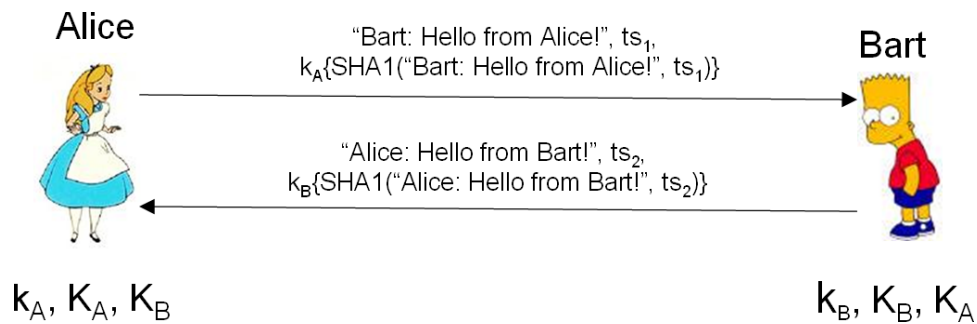


Figure 7: Version 4 of the signing protocol

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?

7 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.