Name: _____ Date:_____

Lab 6 – Practical Cryptography in .NET

Objectives

Part 1 – Generating your Key Pair

Part 2 – Getting the Mission Details

Part 3 – Completing the Mission

Part 4 – Playing with Random Numbers (Extra Credit)

Reflection

Appendix A – ACME Public Key

Appendix B – ACME Database API Documentation

Background/Scenario

Over the past few labs, we have explored a few different methods of network communication to create rich applications with content obtained from external sources. Network communication is becoming almost expected in every application. Although network connectivity brings in dynamic content and makes for a richer application experience, it does come at a cost. The more data you bring in from a third party source, the greater the chance your application can be manipulated if the data has been tampered with before it reaches your application.

The .NET Framework provides many built in classes to help secure your application's communication. In this lab, you will communicate with a web service and explore various methods in secure communication using cryptography in .NET.

```
----BEGIN CONFIDENTIAL MESSAGE----
Welcome, gumshoe to ACME Detective Agency!
As a recent graduate of the ACME Detective Academy, you are now
authorized to access the ACME Detective Database where you will be
receiving your field assignments and mission briefings.
Be warned, all communication to the database needs to be encrypted and
signed otherwise the data could be intercepted and compromised by VILE
agents.  The safety of our detectives is of the utmost importance to us
here at ACME.
This training guide will take you through setting up your employee
access to the database and obtaining your first mission briefing.
This message will not self-destruct but it does have a due date!
----END CONFIDENTIAL MESSAGE----
```

Required Resources

- Visual Studio
- Internet Connection
- Lab 6 Public Key (See Appendix A)

Part 1: Generating your Key Pair

All communication to the ACME Detective Database needs not only to be encrypted but signed as well. Signing the data guarantees that the recipient can verify that the data came from a specific source untampered by validating the signature against the source's public key.

As your first step as a new ACME Detective, you will have to generate your public and private key pair and upload the public key to the ACME Database.

Note: Always keep your private key to yourself. Never distribute your private key to anyone!

Step 1: Generate your keys

C# has several different ways to instantiate an instance of the RSA class. What are the differences?

Initialization	When to use?
RSA.Create()	when you want to create a default instance of the RSA algorithm
new RSACryptoServiceProvider()	to encrypt a string into an array of bytes and then decrypt the bytes back into a string.
new RSACng()	The RSACng class is derived from the RSA class. Therefore, it is not a drop-in replacement for existing uses of RSACryptoServiceProvider.

Use the RSA algorithm to generate a public and private key pair with a 2048 bit length.

Step 2: Export your public and private key pair.

Export out both your public and private keys. If you debug inspect your public key, you will see it has two values, the Exponent and Modulus. The Modulus is your public key value and the Exponent is the public exponent for the key. Serialize and save your private key to disk.

Note: If you serialize the RSAParameters class representation of your private key, by default, the .NET serializers won't serialize the private key portion and will only serialize the public key. You can either serialize all the properties manually or you can use the ToXMLString() function on the RSA class which would generate a full XML representation of the key.

Step 3: Hash the ACME Secret.

Using the SHA256 algorithm, hash the string “lab6”. That hash will be used as the key for the HMAC in the next step.

Step 4: HMAC your public key

Calculating an HMAC of the data that will be transmitted with a shared secret is a good way of ensuring the data is not tampered with during transit. Only the sender and receiver has the ability to calculate the correct HMAC from the data.

Use the HMAC SHA256 algorithm and generate an HMAC hash checksum of the modulus bytes of your public key. Use the SHA256 hash you generated from the previous step as the key for the HMAC. Encode the HMAC to a hexadecimal string.

Step 5: Serialize your data to be uploaded.

The ACME Database upload public key function requires the data to be JSON encoded before uploading. Use a JSON serializer to serialize the data. See the API documentation for JSON property names.

Step 6: Upload your public key

Connect to the ACME Database and upload your public key to the server. See Appendix B for API details.

If successful, you should get your ACME employee ID back from the server. Save your employee ID along with your private key. Your employee ID is now linked to your private key and you will use that ID for the next two parts of the lab.

Note: You only need to upload your public key once. Do not upload your key multiple times. If you call the upload function again, you will get back a new ID.

Part 2: Getting the Mission Details

After uploading your public key to the server, the ACME Database will now be able to send you encrypted messages that you will be able to decrypt using your private key.

Step 1: Request the mission details from the ACME Database.

Request your mission details from the ACME Database. See the Appendix for the API details. Your ID is the ID value that you have received from the previous part. Remember, you don’t need a new ID every time. As long as you have access to your private key and ID, you can call the GetMissionDetails function as many times as you want.

Step 2: Deserialize the Mission Details.

Use a JSON deserializer to deserialze the response from the ACME Database.

Step 3: Verify the signature using RSA.

The signature is for the SHA256 hash checksum of the EncryptedMessage bytes. Generate a SHA256 hash of the EncryptedMessage after converting from a base64 encoded string. Use the ACME public key to verify the data using the RSA algorithm’s verify signature function.

Step 4: Decrypt the AES key using RSA.

Use your private key to decrypt the EncryptedKey using the RSA decryption function.

Step 5: Decrypt the message using AES.

Use the decrypted AES key with the IV to decrypt the EncryptedMessage.

Step 6: Verify the message using SHA256.

Use the SHA256 hashing algorithm and hash the decrypted message. Use the hash result to check to make sure you decrypted the message property by comparing it to the MissionHash value.

Step 7: Read the mission.

If you were successful in decrypting the message from ACME, there should be a Response value that you will use in the next part.

Part 3: Completing the Mission

After reading your mission briefing, you will now need to upload the mission response back to ACME in acknowledgement of the mission.

Step 1: Generate the AES key and IV

Use cryptographic random number generator and generate a 256 bit AES key and IV.

Step 2: Encrypt the mission response using AES.

Using the mission response you got from the previous part, encrypt it with the key you generated in the previous step.

Step 3: Encrypt the AES key using RSA.

Use the ACME public key to encrypt the AES key.

Step 4: Sign the encrypted response using RSA.

Use your private key to sign the encrypted response.

Step 5: Serialize the data to be uploaded.

Refer to the Appendix for the JSON property names for serialization.

Step 6: Upload it to the ACME Database.

Refer to the Appendix for the API details.

Step 7: Read the response.

If you were successful in encrypting and uploading the mission response, you should have got a message back.

What is the answer?

You did it! The answer to the lab question is: chocolate frog

Part 4: Playing with Random Numbers (Extra Credit)

When dealing with encryption it is very important to have random numbers. The System.Random class can generate random enough numbers for general purposes, but for encryption, it should not be used. The System.Random class will always generate the same random numbers every time for a specific seed. For example, the seed 250 will generate the following numbers if you use Next(100): 33 17 51 91 7 3 92 86 90 46…

What is the seed value for this sequence? (Using Next(100))

62 73 79 50 41 86 44 4 12 5

Reflection

1. During this lab, you have used 2048 bit RSA keys, 256 bit AES, and SHA256 for the various parts of the lab. The .NET framework has support to use bigger and smaller bit sizes for the various cryptographic functions. What are the pros and cons for using bigger or smaller bit sizes for these functions?

   for bigger size keys, it will be harder to decrypt, but takes longer to process. for shorter keys, you save time, but these keys because of the smaller bit size are easier to hack. Depending on how much encryption you want to have, you must choose between the bit sized keys.

Appendix A – ACME Detective Agency (Lab 6) Public Key

```
<RSAKeyValue>
      <Modulus>
         tixnDkk0FJbVTFjqVSIqc7YSBHV/QahaM3W0oaU1X1Mlnz3CF2YKmsSzNUdvoW
         zWbFaE3RqMxLgKKJXgyaYsO0SZLijNWPJpaNhjl+HitowBbB3A3WB2vBve0yp2
         azNIYtQ/FsOdRd+e9JM1ltbRQ+dsmjetm6X6XGt2DhUCsupWeQLStyjoHrnH7Q
         keUTM24ks1bd+S/fqozLJnt+jLqkWTJ9om+YNwwwy8oT+VwoXNAva5+IdYTGgp
         LVwfxFKzTIkDOJQMwMY7FQIUc7SMwia+UiPtth8LFOigwp1scbClOm4dE25gu2
         0cpghpnPYnsbISLHOQKUX5oxAViPjzvQ==
      </Modulus>
      <Exponent>AQAB</Exponent>
</RSAKeyValue>
```

Appendix B – ACME Detective Database API Documentation

General API Information

For the previous labs, when accessing web services, you have been using mainly HTTP GET requests. GET requests are primarily for requesting a resource from a web service. The POST method is used to send data to the web service which the server would create or update a resource.

There are other HTTP methods like PUT and DELETE, but the ACME Database doesn't make use of those methods.

If you have been using the WebClient class, the Download methods are all GET requests and the Upload methods are all POST.

When passing parameters to a GET method, you typically would append all the parameters at the end of the base URL with a question mark character, then the parameter name, then an equal symbol, then the value of the parameter. If you have multiple parameters to specify, add an ampersand character between the parameter value pairs. For example:

```
GET http://cst407.azurewebsites.net/Lab6/{function}?{p1}={d1}&{p2}={d2}
```

Passing parameters for POST requests requires a little more preparation. The ACME Database requires all POST requests to be JSON formatted. To do so, you will have to create an object with properties that match the parameters requested by the endpoint and serialize it to JSON. Also, you may need to specify the header Content-Type and set it to application/json to let the server know you are sending JSON data. If using the WebClient class, use the Headers property and call the Add function to pass the header.

```
POST http://cst407.azurewebsites.net/Lab6/{function}
Header:
Content-Type: application/json
String Data to POST:
{
  parameter1: data1,
  parameter2: data2
}
```

All API calls will return a JSON object with two fields, Status and Data.

```
{
  "Status": "OK",
  "Data": {...}
}
```

The Status field would either be OK or ERROR.

If OK, the Data field would return another JSON object of the data you requested.

If ERROR, the Data field would be null and there will be a new Message field string response explaining why it returned error.

```
{
  "Status": "ERROR",
  "Message": "The error message."
}
```

Uploading a Public Key

```
POST http://cst407.azurewebsites.net/Lab6/UploadPubKey
```

Parameters	What it does
key	A base64 encoded string representation of the byte[] of your RSA public key. (The modulus parameter)
exp	A base64 encoded string representation of the byte[] of your RSA public key exponent. In most cases, it will probably be “AQAB”
secret	A SHA256 HMAC hex string of your RSA public key bytes using the SHA256 hash representation of “lab6” as the key for the HMAC.

Return Data

Value	Description
ID	An integer value of your ACME employee ID. Save this value! You will use this ID as a parameter for all other functions in this lab.

Note: You should only need to ever call this method once. Once you have your ID, you will use that for the lifetime of the lab.

Getting Mission Data

```
GET http://cst407.azurewebsites.net/Lab6/GetMissionDetails
```

Parameters	What it does
id	Your ACME employee ID that you received from uploading your public key.

Return Data

Value	Description
EncryptedMessage	The base64 representation of the encrypted mission message bytes. The encryption is AES.
EncryptedKey	The base64 representation of the AES key used to encrypt the message. This value is encrypted using your uploaded public key.
MissionHash	The SHA256 hash of the decrypted message in hexadecimal. Use this value to verify that you successfully decrypted the message.
IV	The IV used in the AES encryption process.
Signature	The signature of the SHA256 hash checksum of the EncryptedMessage bytes using ACME's private key in base64. Verify the signature using the public key attached to the lab. (Appendix A)

Uploading Mission Response

```
POST http://cst407.azurewebsites.net/Lab6/FinishMission
```

Parameters	What it does
id	Your ACME employee ID that you received from uploading your public key.
key	The base64 representation of the AES key used to encrypt the message. Generate using the cryptographic random number generator. This value is encrypted using ACME's public key. See Appendix A.
iv	The base64 representation of the IV used for the AES encryption. Generate using the cryptographic random number generator.
msg	The base64 representation of the encrypted message. The encryption method is AES.
sig	The signature of the encrypted message using your private key.

Return Data

Value	Description
Response	The response to the completed mission. Used to answer the question in the lab.