Homework 1
Cryptography

This homework requires the use of OpenSSL, which is available for Linux, Mac OS X, and Windows. Students can SSH to a Linux machine to use OpenSSL (e.g., csgrads1.utdallas.edu or cs1.utdallas.edu), and use the Secure File Transfer Protocol (SFTP) to transfer local files to/from the UT Dallas Linux machine. See: https://www.utdallas.edu/oit/howto/use-sftp/

1. Symmetric Encryption 10%

Download file named **encrypted** from eLearning. This file is encrypted with key size of 128 bits using the cbc mode: i.e., **aes-128-cbc** cipher mode of **openssl.**

a) Decrypt this file using the following information:

secret key:
00112233445566778899887766554

Initialization Vector (IV):
143124152

b) Write down in your report the command you used for decrypting the file.

c) Write down the message contained in the decrypted file.

2. Hash Functions 10%

a) Using **openssl**, write in your report the SHA-1 digest of the **encrypted** file.

b) Write in your report the SHA-256 digest of the **encrypted** file.

c) Go to the website https://shattered.io and read the description of the problem. Summarize in a couple of paragraphs what the researchers found and why it is important.

3. Modes of Encryption 20%

We will encrypt a photo of Temoc under various modes of encryption. You have the option to use **openssl** or the application **LabBlockCipherOperatingModes.jar** included in eLearning:

Using OpenSSL:
(I highly encourage you to use this, so you can learn more about how to encrypt image files):

```
1. Encrypt temoc.bmp (in this example we use ECB mode with key = 0011223344556677)
$ openssl enc -aes-128-ecb -K 0011223344556677 -in temoc.bmp -out temoc-ecb.bmp
```

Now, you will notice that you cannot open the file **temoc-ecb.bmp**. The reason for this is that you encrypted the entire file including the header file that tells this file is a bitmap image file. To fix this, we need to modify the binary file by copying the file header (for **temoc.bmp**, it is the first 122 bytes of the file) from the original file to the encrypted one. We can do this by using a "hex editor" such as "Hex Fiend"; or if we are running Mac or Linux we can use the "dd utility":

```
2. Replace the encrypted header with the header from the original file
$ dd if=temoc.bmp of=temoc-ecb.bmp bs=122 count=1 conv=notrunc
```

Now the encrypted file **temoc-ecb.bmp** will be treated as a legitimate **.bmp** file.

Or, using LabBlockCipherOperatingModes.jar:
Open the file **LabBlockCipherOperatingModes.jar** and load **temoc.bmp**. You can encrypt the image by interacting with the application's user interface.

Now that you know how to encrypt image files, do the following:
a) Encrypt **temoc.bmp** with ECB mode (e.g., use **aes-128-ecb** in **openssl**). Take a screenshot.

b) What happened? Explain the problem with ECB mode.

c) Now encrypt with CBC or CFB mode (e.g., **aes-128-cbc** or **aes-128-cfb**). Take screenshots.

d) Look up CBC, PCBC, and CFB modes of encryption. Explain briefly them.

e) Now encrypt the image with OFB mode (use **des-ofb** in **openssl**), with key and initialization vector (IV) in zeros. Take screenshots. Change the initialization vector but keep the key in zeroes and look at the new results. Explain why this happens. Briefly explain what is an IV.

f) Nowadays most encryption algorithms also offer authentication, and this is known as authenticated encryption. Look online for authenticated encryption modes. Write down what mode of encryption is your favorite and explain why you made this decision.

4. Signatures 20%

This question is based on Lecture 5 slide 24 where we learned about digital signatures using public key crypto. We saw that we can calculate the signature of a message using: **`signature=sign(sk, m)`** and we can verify the signature with: **`verify(pk, signature, m)`**.

a) In practice, we sign the hash of the message (e.g., **`sign(sk, H(m))`**), not the message. Why?

Now, download the files **`message`**, **`signature`**, **`public_key_1.pem`**, and **`public_key_2.pem`**.

The hash of file **`message`** was digitally signed using the private key of either Alice or Bob, and the digital signature was saved as **`signature`**. We used the following command for this step:

```
$ openssl dgst -sha256 -sign <PRIVATE_KEY>.pem -out signature message
```

To verify the signature we use:

```
$ openssl dgst -sha256 -verify <PUBLIC_KEY>.pem -signature signature message
```

b) Let's assume **`public_key_1.pem`** belongs to Alice and **`public_key_2.pem`** belongs to Bob. Can you guess who signed? Alice or Bob? Explain why, and take a screenshot of your results.

5. "Vanilla" RSA Encryption 20%

You are using the "Vanilla" version of RSA for encryption/decryption as illustrated in Lecture 9.

Your primes are **p=47** and **q=71**, and your public key is **N=3337** and **e=79**.

a) Which of these numbers is your private key **d**? 79, 128, 1019, or 2059?
(hint: we can rewrite $d * e \equiv 1 \bmod \phi(N)$ that we saw in class as $(d * e) \bmod \phi(N) = 1$)

b) You receive the following encrypted message: **2423**. Find the decryption. Show your work.

Note:
You may want to use Google Calculator (just search for "calculator" on Google) and you may also want to check algorithms in Khan Academy for calculating fast modular exponentiation: https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic/p/fast-modular-exponentiation

Or, use Wolfram Alpha: https://www.wolframalpha.com

6. "Vanilla" RSA Program 10%

Run the program **`LabRSAStepByStep.jar`**.

The algorithm starts with two big primes **p** and **q**. These primes are generated such that **p-1** and **q-1** are relative primes to the encryption exponent, which has been chosen here as **65537**.

a) What is the definition of a relative prime?

Click on the buttons labelled "Generate **p**" and "Generate **q**". Make sure they are not the same. Then click on the "Next >" button.

The last step of key generation is calculating a **d** such that **e * d** is congruent modulo **(p-1)(q-1)** to **1**. Click on "Next >" and then "Calculate" to see the decryption key "**d**". This number **d** is called the "modular inverse" of **e** and is calculated using the "Extended Euclidean Algorithm".

Click on the "Next >" button. Now that you got a key pair, we will try encrypting some data. For now, we will encrypt only long decimal numbers, however you know that any message or file can be represented as them.

Type a number (lower than **n**) in the box labelled "With public key" and then click on the encryption equation.

b) Take a screenshot of this result.

c) The encryption appears at the bottom of the box. Copy the encryption and paste it in the box labelled "With private key". Then click on the button representing the decryption operation.

d) Take a screenshot of this result. Explain what happened.

e) Vanilla RSA is not secure (similar as to why the ECB mode is not secure). In practice, we use RSA-OAEP. Lookup what RSA-OAEP stands for and explain briefly its operation.

7. Crypto Sabotage 10%

These questions are based on the paper included in the homework files:

a) For each example of historical sabotage identify:

- Who was responsible?
- What was the technical implementation of the sabotage?
- What was the consequence to the operation of the overall system?

b) How can a saboteur expose the internal state of pseudorandom number generators?

c) Why is exposing the internal state of pseudorandom number generators a threat to TLS?