UNIVERSITY OF MARYLAND
DEPARTMENT OF ELECTRICAL AND COMPUTER ENGINEERING

ENEE 459C
Computer Security
Instructor: Charalampos Papamanthou

Homework 3

Out: 10/26/15 Due: 11/06/15

Instructions

1. Strictly adhere to the University of Maryland Code of Academic Integrity.
2. **Type** and submit your solutions as a pdf document at Canvas. Include your full name in the solutions document. Name the solutions document as x-hw3.pdf, where x is your last name.
3. No extensions will be given. Everything will be due at 11.59pm on 11/06/15.

Problem 1 (25 points)

1. You are given that $3 \cdot 5 - 2 \cdot 7 = 1$. Use the Chinese remainder theorem to compute
$$3^{72} \mod 35 .$$
2. The Blum-Micali generator outputs the next pseudorandom bit as follows: First it computes
$$x_i = g^{x_{i-1}} \mod p$$
and then it outputs the bit 1 if $x_i < \frac{p-1}{2}$, else it outputs the bit 0 (the procedure is repeated for $i = 0, 1, 2, \ldots$). Write a program that implements `get_random_bit()` using the above generator for $p = 2147483647$ (note that in real applications, p needs to be a lot bigger) and $g = 7$. Use $x_0 = 1$. Also, write a program that implements `get_random_number_BM()` and outputs a 32-bit pseudorandom number by calling `get_random_bit()` 32 times. Note that for implementing the exponentiation above, you should use the modular power algorithm (that runs in logarithmic time) that we discussed in class.

Problem 2 (25 points)

1. What is the difference between public-key encryption and secret-key encryption? What is a certification authority? What is a digital certificate?
2. Suppose Mallory is responsible for storing all the public keys of the students that are currently enrolled at the University of Maryland. Namely, it keeps a list of mappings of (d_i, pk_i), where d_i is the directory identifier of student i and pk_i is the public key of student i. Suppose Bob, Alice and Eve are all students at the university. Bob wants to send a confidential message to Alice and asks Mallory for Alice's public key. However Eve is really interested in learning what Bob wants to say to Alice. Eve happens to be very good friends with Mallory. Explain how Eve could collaborate with Mallory so that Eve eventually gets to decrypt Bob's message to Alice.

3. Instead of using Mallory for managing public keys, Bob and Alice decide to setup their public keys with Verisign, Inc., a trusted certification authority. In order to arrange a meeting for the weekend, Bob exchanges with Alice brief text messages, using RSA encryption. Let p_B and p_A be the public keys of Bob and Alice respectively. A message m sent by Bob to Alice is transmitted as $\mathsf{Enc}_{p_A}(m)$ and the reply r from Alice to Bob is transmitted as $\mathsf{Enc}_{p_B}(r)$. Although Eve cannot collaborate with Mallory anymore, she can still eavesdrop the communication and knows the following information:
   - Bob and Alice are meeting either on a Saturday or on a Sunday;
   - The set of 100 possible meeting places;
   - Alice is going to specify the time in the format HH:MM pm or HH:MM am.
   - Messages and replies are terse exchanges of the following form:

     Bob: When are we meeting?
     Alice: Saturday.
     Bob: Where are we meeting?
     Alice: Union station.
     Bob: What time?
     Alice: 8.30pm.

   Describe how Eve can learn the meeting day, time, and place of Bob and Alice.
4. As a result of the above attack, Bob and Alice decide to modify the protocol for exchanging messages. Describe two simple modifications of the protocol that are not subject to the above attack. The first one should use random numbers and the second one should use symmetric encryption.

Problem 3 (25 points)

In class we talked about some basic number theory and its significance for various cryptographic algorithms such as RSA and ElGamal. Answer the following questions:

1. Suppose $n = 43434563$. How many elements in $\mathbb{Z}_n - \{0\}$ have a multiplicative inverse modulo n? Name such an element (i.e., one that has a multiplicative inverse modulo n). You might want to visit this webpage to answer this question.
2. Consider an RSA key set with $p = 11$, $q = 29$, $n = pq = 319$, and $e = 3$. What value of d should be used in the secret key? What is the encryption of message $M = 100$? Why should e always be an odd number (to answer this question, assume p and q are always greater than 2)? You might want to visit this webpage to answer this question.
3. Prove that *all* elements in $\mathbb{Z}_n - \{0\}$ have a multiplicative inverse modulo n **if and only if** n is a prime. Note that you need to prove both directions.
4. Let now n be a positive integer and let $\mathbb{Z}_n^* = \{x \in [0, n] : \mathsf{GCD}(x, n) = 1\}$. Let also $k < n$ and $a \in \mathbb{Z}_n^*$. What is the multiplicative inverse of $a^k \in \mathbb{Z}_n^*$ when n is a prime? What is the multiplicative inverse of $a^k \in \mathbb{Z}_n^*$ in case that n is not a prime (and has p and q as prime factors)? You do not have to use the extended Euclidean algorithm to answer this question.
5. Prove that for all integers a, k and n, it is $\mathsf{GCD}(a, n) = \mathsf{GCD}(a + kn, n)$ (Hint: Use the same technique we used in class to prove that $\mathsf{GCD}(a, b) = \mathsf{GCD}(b, a \bmod b)$).
6. Prove that for all integers a, b and primes p it is $(a + b)^p = a^p + b^p \mod p$ (Hint: Use Fermat's little theorem).

7. Recall the extended Euclidean algorithm: Given two integers a and b, it outputs two other integers x and y such that $xa + yb = \mathsf{GCD}(a, b)$. Devise an algorithm that uses the extended Euclidean algorithm as a black box, such that, on input 3 integers a, b and c, it outputs three integers u, v and z such that $ua + vb + zc = \mathsf{GCD}(a, b, c)$ (Hint: Use the fact that $\mathsf{GCD}(a, b, c) = \mathsf{GCD}(\mathsf{GCD}(a, b), c)$).

Problem 4 (25 points—*taken from SEED labs* `http://www.cis.syr.edu/~wedu/seed/`)

For this lab assignment, you will use the virtual machine that you have downloaded from SEED labs. Please provide all the screenshots and the code in your final report.

The learning objective of this lab is for students to get familiar with the concepts in the Public-Key encryption and Public-Key Infrastructure (PKI). After finishing the lab, students should be able to gain a first-hand experience on public-key encryption, digital signature, public-key certificate, certificate authority, authentication based on PKI. Moreover, students will be able to use tools and write programs to create secure channels using PKI.

Task 1: Become a Certificate Authority (CA)

A Certificate Authority (CA) is a trusted entity that issues digital certificates. The digital certificate certifies the ownership of a public key by the named subject of the certificate. A number of commercial CAs are treated as root CAs; VeriSign is the largest CA at the time of writing. Users who want to get digital certificates issued by the commercial CAs need to pay those CAs.

In this lab, we need to create digital certificates, but we are not going to pay any commercial CA. We will become a root CA ourselves, and then use this CA to issue certificate for others (e.g. servers). In this task, we will make ourselves a root CA, and generate a certificate for this CA. Unlike other certificates, which are usually signed by another CA, the root CA's certificates are self-signed. Root CA's certificates are usually pre-loaded into most operating systems, web browsers, and other software that rely on PKI. Root CA's certificates are unconditionally trusted.

The Configuration File `openssl.conf`**.** In order to use `OpenSSL` to create certificates, you have to have a configuration file. The configuration file usually has an extension `.cnf`. It is used by three `OpenSSL` commands: `ca`, `req` and `x509`. The manual page of `openssl.conf` can be found using Google search. You can also get a copy of the configuration file from `/usr/lib/ssl/openssl.cnf`. After copying this file into your current directory, you need to create several sub-directories as specified in the configuration file (look at the `[CA_default]` section):

```
dir             = ./demoCA          # Where everything is kept
certs           = $dir/certs        # Where the issued certs are kept
crl_dir         = $dir/crl          # Where the issued crl are kept
new_certs_dir   = $dir/newcerts     # default place for new certs.

database        = $dir/index.txt    # database index file.
serial          = $dir/serial       # The current serial number
```

For the `index.txt` file, simply create an empty file. For the `serial` file, put a single number in string format (e.g. 1000) in the file. Once you have set up the configuration file `openssl.cnf`, you can create and issue certificates.

Certificate Authority (CA). As we described before, we need to generate a self-signed certificate for our CA. This means that this CA is totally trusted, and its certificate will serve as the root certificate. You can run the following command to generate the self-signed certificate for the CA:

```
$ openssl req -new -x509 -keyout ca.key -out ca.crt -config openssl.cnf
```

You will be prompted for information and a password. Do not lose this password, because you will have to type the passphrase each time you want to use this CA to sign certificates for others. You will also be asked to fill in some information, such as the Country Name, Common Name, etc. The output of the command are stored in two files: `ca.key` and `ca.crt`. The file `ca.key` contains the CA's private key, while `ca.crt` contains the public-key certificate.

Task 2: Create a Certificate for `ENEE459CServer.com`

Now, we become a root CA, we are ready to sign digital certificates for our customers. Our first customer is a company called `ENEE459CServer.com`. For this company to get a digital certificate from a CA, it needs to go through three steps.

Step 1: Generate public/private key pair. The company needs to first create its own public/private key pair. We can run the following command to generate an RSA key pair (both private and public keys). You will also be required to provide a password to encrypt the private key (using the AES-128 encryption algorithm, as is specified in the command option). The keys will be stored in the file `server.key`:

```
$ openssl genrsa -aes128 -out server.key 1024
```

The `server.key` is an encoded text file (also encrypted), so you will not be able to see the actual content, such as the modulus, private exponents, etc. To see those, you can run the following command:

```
$ openssl rsa -in server.key -text
```

Step 2: Generate a Certificate Signing Request (CSR). Once the company has the key file, it should generates a Certificate Signing Request (CSR), which basically includes the company's public key. The CSR will be sent to the CA, who will generate a certificate for the key (usually after ensuring that identity information in the CSR matches with the server's true identity). Please use `ENEE459CServer.com` as the common name of the certificate request.

```
$ openssl req -new -key server.key -out server.csr -config openssl.cnf
```

It should be noted that the above command is quite similar to the one we used in creating the self-signed certificate for the CA. The only difference is the `-x509` option. Without it, the command generates a request; with it, the command generates a self-signed certificate.

Step 3: Generating Certificates. The CSR file needs to have the CA's signature to form a certificate. In the real world, the CSR files are usually sent to a trusted CA for their signature. In this lab, we will use our own trusted CA to generate certificates. The following command turns the certificate signing request (`server.csr`) into an X509 certificate (`server.crt`), using the CA's `ca.crt` and `ca.key`:

```
$ openssl ca -in server.csr -out server.crt -cert ca.crt -keyfile ca.key \
             -config openssl.cnf
```

If OpenSSL refuses to generate certificates, it is very likely that the names in your requests do not match with those of CA. The matching rules are specified in the configuration file (look at the [policy_match] section). You can change the names of your requests to comply with the policy, or you can change the policy. The configuration file also includes another policy (called policy_anything), which is less restrictive. You can choose that policy by changing the following line:

```
"policy = policy_match"  change to "policy = policy_anything".
```

Task 3: Use PKI for Web Sites

In this lab, we will explore how public-key certificates are used by web sites to secure web browsing. First, we need to get our domain name. Let us use ENEE459CServer.com as our domain name. To get our computers recognize this domain name, let us add the following entry to /etc/hosts; this entry basically maps the domain name ENEE459CServer.com to our localhost (i.e., 127.0.0.1):

```
127.0.0.1  ENEE459CServer.com
```

Next, let us launch a simple web server with the certificate generated in the previous task. OpenSSL allows us to start a simple web server using the s_server command:

```
# Combine the secret key and certificate into one file
% cp server.key server.pem
% cat server.crt >> server.pem

# Launch the web server using server.pem
% openssl s_server -cert server.pem -www
```

By default, the server will listen on port 4433. You can alter that using the -accept option. Now, you can access the server using the following URL: https://ENEE459CServer.com:4433/. Most likely, you will get an error message from the browser. In Firefox, you will see a message like the following: *"enee459cserver.com:4433 uses an invalid security certificate. The certificate is not trusted because the issuer certificate is unknown".*

Had this certificate been assigned by VeriSign, we will not have such an error message, because VeriSign's certificate is very likely preloaded into Firefox's certificate repository already. Unfortunately, the certificate of ENEE459CServer.com is signed by our own CA (i.e., using ca.crt), and this CA is not recognized by Firefox. There are two ways to get Firefox to accept our CA's self-signed certificate.

- We can request Mozilla to include our CA's certificate in its Firefox software, so everybody using Firefox can recognize our CA. This is how the real CAs, such as VeriSign, get their certificates into Firefox. Unfortunately, our own CA does not have a large enough market for Mozilla to include our certificate, so we will not pursue this direction.
- **Load** ca.crt **into Firefox:** We can manually add our CA's certificate to the Firefox browser by clicking the following menu sequence:

  ```
  Edit -> Preference -> Advanced -> View Certificates.
  ```

  You will see a list of certificates that are already accepted by Firefox. From here, we can "import" our own certificate. Please import ca.crt, and select the following option: "Trust this CA to identify web sites". You will see that our CA's certificate is now in Firefox's list of the accepted certificates.

Now, point the browser to `https://ENEE459CServer.com:4433`. Please describe and explain your observations. Please also do the following tasks:

1. Modify a single byte of `server.pem`, and restart the server, and reload the URL. What do you observe? Make sure you restore the original `server.pem` afterward. Note: the server may not be able to restart if certain places of `server.pem` is corrupted; in that case, choose another place to modify.
2. Since `ENEE459CServer.com` points to the localhost, if we use `https://localhost:4433` instead, we will be connecting to the same web server. Please do so, describe and explain your observations.

Task 4: Establishing a TLS/SSL connection with server

In this task, we will implement a TCP client and TCP server, which are connected via a secure TCP connection. Namely, the traffic between the client and the server are encrypted using a session key that are known only to the client and the server. Moreover, the client needs to ensure that it is talking to the intended server (we use `ENEE459CServer.com` as the intended server), not a spoofed one; namely, the client needs to authenticate the server. This server authentication should be done using public-key certificates[1].

`OpenSSL` has implemented the SSL protocol that can be used to achieve the above goals. You can use `OpenSSL`'s SSL functions directly to make an SSL connection between the client and the server, in which case, the verification of certificates will be automatically carried out by the SSL functions. There are many online tutorials on these SSL functions, so we will not give another one here. Several tutorials are linked in the web page of this lab.

We provide two example programs, `cli.cpp` and `serv.cpp`, in a file `demo_openssl_api.zip`, to help you to understand how to use `OpenSSL` API to build secure TCP connections. The file can be downloaded from the lab's web page. The programs demonstrate how to make SSL connections, how to get peer's certificate, how to verify certificates, how to get information out of certificates, etc. To make the program work, you have to unzip it first and run the `make` command. The zip file includes a certificate for server and another for the client. The passwords (private keys are encrypted using the passwords) are included in the README file.

Tasks. Using the provided example as your basis, you should do the following tasks and describe your activities, observations, and answers in your lab report:

- Please use the server certificate that you generated in Task 2 as the certificate for the server.
- The client program needs to verify the server certificate. The verification consists of several checks. Please show where each check is conducted in your code (i.e., which line of your code does the corresponding check):
  1. The effective date
  2. Whether the server certificate is signed by an authorized CA
  3. Whether the certificate belongs to the server
  4. Whether the server is indeed the machine that the client wants to talk to (as opposed to a spoofed machine).

[1] In practice, the server also needs to authenticate the client. However, for the sake of simplicity, we do not require client authentication in this task.

To answer this question using your first-hand experience, you can modify the server's certificate and private key, the CA's certificate, etc.; you can then run your program, and see which line of your code reports errors.

- The provided sample code for the server also verifies the client's certificate. We do not need this, please remove this part of code, and show us what changes you made in the server-side code.
- What part of the code is responsible for the key exchange, i.e. for both sides to agree upon a secret key?

Note: To find out where the effective date is checked, you can either create a certificate that has an invalid effective date, or you can change your system time. You can use the following command to do so:

```
% sudo date --set="1 May 2000"
```

It should be noted that within a few seconds, the date will be set back to the correct date due to the time synchronization service running on the system. You can either disable that service using the following command, or simply conduct the experiment within the very short time window. If you stop the service, make sure you restart it after your experiment, or the timestamps in your screenshots will not be the current time, and your lab reports may end up being rejected by your instructor.

```
Disable the time synchronization service
% sudo service vboxadd-service stop

Restart the time synchronization service
% sudo service vboxadd-service start
```

Task 5: Performance Comparison: RSA versus AES

In this task, we will study the performance of public-key algorithms. Please prepare a file (`message.txt`) that contains a 16-byte message. Please also generate an 1024-bit RSA public/private key pair. Then, do the following:

1. Encrypt `message.txt` using the public key; save the the output in `message_enc.txt`.
2. Decrypt `message_enc.txt` using the private key.
3. Encrypt `message.txt` using a 128-bit AES key.
4. Compare the time spent on each of the above operations, and describe your observations. If an operation is too fast, you may want to repeat it for many times, and then take an average.

After you finish the above exercise, you can now use `OpenSSL`'s `speed` command to do such a benchmarking. Please describe whether your observations are similar to those from the outputs of the `speed` command. The following command shows examples of using `speed` to benchmark `rsa` and `aes`:

```
% openssl speed rsa
% openssl speed aes
```

Task 6: Create Digital Signature

In this task, we will use `OpenSSL` to generate digital signatures. Please prepare a file (`example.txt`) of any size. Please also prepare an RSA public/private key pair. Do the following:

1. Sign the SHA256 hash of `example.txt`; save the output in `example.sha256`.
2. Verify the digital signature in `example.sha256`.
3. Slightly modify `example.txt`, and verify the digital signature again.

Please describe how you did the above operations (e.g., what commands do you use, etc.). Explain your observations. Please also explain why digital signatures are useful.

Hint: Check openssl command line tool for more details. Use Bless Hex Editor (pre-installed in your virtual machine) to modify the documents. Also, check some sample code and helpful links here: `http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Crypto/Crypto_PublicKey/`