Emmanuel Taylor

October 17, 2014

CMSC414 0201

Dr. Shankar

**Project 2: Crypto Lab – Public Key Cryptography and PKI**

**Purpose:**

The purpose of this lab is to become familiar with the concepts that are used in Public Key Encryption/Decryption and Public Key Infrastructure (PKI). By completing this lab, I gained first-hand experience on public-key encryption, digital signatures, public-key certificates, certificate authority, and authentication based on PKI. I also learned how to write an efficient bash shell script.

**Task 1: Become a Certificate Authority (CA)**

In this task, we became a Certificate Authority. A Certificate Authority is a trusted entity that issues digital certificates which certifies the ownership of a public key. I learned how to become a root CA and generates certificates as well. Because I was the root CA, my certificates were self-signed. I began this task by obtaining a copy of the OpenSSL configuration file (openssl.cnf). After copying this file into my directory, I created the sub-directories that were specified in the configuration file. These directories are shown below:

dir = ./demoCA # Where everything is kept

certs = $dir/certs # Where the 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 file, I created an empty text file and for the serial file, I put a single number in string format. The number used was 1000. Next, we generated a self-signed certificate for our CA in order to ensure that it is trusted. This was accomplished by using the following command.

openssl req –new –x509 –keyout ca.key –out ca.crt –config openssl.cnf

After using this command and entering all of the requested information, we chose a PEM Pass Phrase which I set to be “powerball1” (without quotes). The output of the command was stored in two files named ca.key and ca.crt. The ca.key file contains the private key while ca.crt contains the public key certificate.

**Task 2: Create a Certificate for PKILabServer.com**

For this task, we signed a digital certificate for a company called PKILabServer.com. For this company to get a digital certificate, we performed three steps. The first step was to generate a public/private key pair. This can be accomplished by using the following command which generates a 1024 bit RSA public/private key pair:

openssl genrsa –des3 –out server.key 1024

The –des3 tag allows us to password protect our keys. The password I chose for server.key was “powerball2” (without quotes). The second step was to generate a certificate signing request or CSR. The CSR is used to ensure that the identity information matches with the server’s true identity. This was achieved by using the following command.

openssl req –new –key server.key –out server.csr –config openssl.cnf

This allows us to generate a certificate for server.key. The name “PKILabServer.com” was used as our common name for this task. The final step was to generate certificates. This can be achieved by using the following OpenSSL command:

openssl ca –in server.csr –out server.crt –cert ca.cry –keyfile ca.key –config openssl.cnf

This command allows us to sign our own certificates. While attempting this step, OpenSSL refused to generate certificates because the matching rules were too strict. In order to resolve this problem, I simply edited the policy by changing it from policy = policy\_match to policy = policy\_anything.

**Task 3: Use PKI for Web Sites**

In this task, we learned how public-key certificates are used by web sites to secure web browsing. To test this out, we added PKILabServer.com to our localhost as a domain name by adding 127.0.0.1 PKILabServer.com to /etc/hosts. Next, we launched a simple web server with the certificate that we generated in the previous task. This was accomplished by using the following OpenSSL commands:

# Combing 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

As mentioned in the lab manual, by default, the server listens on port 4433. We accessed the server by following the URL: <https://PKILabServer.com:4433/> which gave us a warning that the certificate was not yet trusted. To fix this problem, we manually added our CA to Mozilla Firefox. After allowing our CA to be trusted, I attempted to connect to the server again by following the same URL. What I observed were several lists including Ciphers supported in s\_server binary, Ciphers common between both SSL end points, and the current session in which 2 server accepts were present and 1 server accept finished. Screenshots 1 and 2 show the full contents of the web page. Using the ghex2 tool in bash, I modified a single byte of server.pem and attempted to follow the URL again. For the most part, the contents of the page were mostly the same accept I did notice that there were only 1 server accepts as opposed to 2. Screenshot 3 shows this difference. Recall that PKILabServer.com points to the localhost. By following <https://localhost:4433/> I was able to connect to the same server, however, I had to make sure my CA was trusted again. Screenshot 4 shows the error message that appeared.

**Task 4: Using PKI to establish secure TCP connections with PKILabServer.com**

In this task, we implemented a TCP client and a TCP server which are connected by a secure TCP connection. The client needs to ensure that it is talking to the intended server, PKILabServer.com in our case, and not a spoofed one. This authentication is done with public-key certificates. Two examples programs, cli.cpp and serv.cpp, were provided to us and did the majority of the dirty work. In order to complete this task, I first repeated Task 2 for the client. I generate a new public/private key pair for the client. This can be accomplished by using the following command which generates a 1024 bit RSA public/private key pair:

openssl genrsa –des3 –out client.key 1024

The –des3 tag allows us to password protect our keys. The password I chose for server.key was “powerball5” (without quotes). The second step was to generate a certificate signing request or CSR. The CSR is used to ensure that the identity information matches with the client’s true identity. This was achieved by using the following command.

openssl req –new –key client.key –out client.csr –config openssl.cnf

This allows us to generate a certificate for client.key. The name “PKILabServer.com” was used as our common name for this task. The final step was to generate certificates. This can be achieved by using the following OpenSSL command:

openssl ca –in client.csr –out client.crt –cert ca.cry –keyfile ca.key –config openssl.cnf

This command allows us to sign our own certificates. While attempting this step, OpenSSL refused to generate certificates because the matching rules were too strict. In order to resolve this problem, I simply edited the policy by changing it from policy = policy\_match to policy = policy\_anything.

After this, I made copies of cli.cpp, serve.cpp, and the Makefile and tailored them to use my CA, server keys, and client keys. After running make, I first ran the server executable which prompted me for my server.key password which is “powerball2” (no quotes). After this, I opened a new terminal and ran the client executable which prompted me for my client.key password which is “powerball5” (no quotes). Upon doing this, my certificate information was authenticated and displayed on the terminal. A screenshot of both the client and server terminals is included in the zip file.

**Task 5: Performance Comparison: RSA versus AES**

In this task, we compared the speed of both the RSA and AES public-key algorithms. We created a file named message.txt that contained a 16-byte message. Next I generated a 1024 bit RSA public/private key pair and encrypted message.txt and then decrypted it right after. I did the same thing for the AES algorithm by using a 128 bit AES key. This task was completed both within a bash shell script in which I performed the aforementioned tasks 50 times and took the average time for each algorithm and in two C programs. According to my performance testing, the AES algorithm is faster than the RSA algorithm and this is consistent with OpenSSL’s speed commands. A copy of the shell script can be found in this zip archive and is titled “enc\_speed.sh.” To run the shell script, simply type in “bash enc\_speed.sh”. To run the C executables, type “./rsa or ./aes”.

**Task 6: Create Digital Signature**

In this task, we used OpenSSL to generate digital signatures. To do this, we created a file named example.txt and provided an RSA public/private key pair to encrypt/decrypt with. Next, we signed the SHA256 hash of example.txt into an output named example.sha256. After, we verified the digital signature in example.sha256 and finally, we slightly modified example.txt and verified the digital signature again. These tasks were completed within a bash shell script titled, “digital\_signature.sh.” As you can see by the displayed output, modifying example.txt changes the digital signature.