Crypto Lab – One Way Hash and Public Key Encryption

CS5460 Assignment 4 Due: October 11 2017

Austin Derbique A01967241

NOTE For this lab, the pass phrase used in all encryptions unless otherwise stated is “1234567890”

3.One-Way Hash Lab

3.1 Task 1: Generating Message Digest and MAC

For this task, I used file.txt in directory part3/task1 and used various digest types piped to respective files. These are the results:

MD5(file.txt)= f14882493b0fa1029a262104653eeb40

SHA1(file.txt)= 0910445d87bda7c321c7328c6e5cf1372973c26c

SHA256(file.txt)= 2611f3e1ac900fcecf3dafb0b40d5f974e12816f849c94b09720b59e1d538d56

The SHA256 hashing digest is longer than the others. Additionally, SHA1 is longer than MD5.

3.2 Task 2: Keyed Hash and HMAC

located inside of part3/task2

SHORT HMAC-MD5(file.txt)= e583d79a56746f06bc315cdb5136c2f3

LONG HMAC-MD5(file.txt)= e583d79a56746f06bc315cdb5136c2f3

SHORT HMAC-SHA1(file.txt)= 4b8c77cf343eb0ae38e280394777fc2c4302423a

LONG HMAC-SHA1(file.txt)= 4b8c77cf343eb0ae38e280394777fc2c4302423a

SHORT HMAC-SHA256(file.txt)=
fe2211327616452af9f8b7e7d360c9aff752ead7deb57a768340e3903f7b62ab

LONG HMAC-SHA256(file.txt)=
fe2211327616452af9f8b7e7d360c9aff752ead7deb57a768340e3903f7b62ab

In this test, I gave the short digest a key of “abcdefg” and the long digest a key of “abcdefghijklmnopqrstuvwxyz”. Interestingly enough, the has comes out to be the same. This makes me believe that only a first set of bytes are used as the key. So In this case, the key size for HMAC doe not matter as only a certain amount of bytes from that key are used.

3.3 Task 3: The Randomness of One-way Hash

Located inside of part3/task3

ORIG MD5(file.txt)= f14882493b0fa1029a262104653eeb40

MOD MD5(file.txt)= bb66183223dfb49f0d279f53433f289c

ORIG SHA256(file.txt)= 2611f3e1ac900fcecf3dafb0b40d5f974e12816f849c94b09720b59e1d538d56

MOD SHA256(file.txt)= 2b9b1a19a2f987a96880582acc209aa1bff7612eaccd08bced34a128769f62b9

It is observed that after flipping one bit using bless, the hashes come out completely different.

4. Public Key Cryptography Lab

4.1 Task 1: Become a Certificate Authority (CA)

Task is complete. Please look at part4/task1/demoCA

4.2 Task 2: Create a Certificate for PKILabServer.com

Task is complete. Please look at part4/task/1/demoCA. The decrypted public/private key pair is named decrypted_server.key.txt. The certificate is signed and generated. Details are also available at generated_certificate_proof.txt.

4.3 Task 3: Use PKI for Web Sites

Before importing the certificate, I attempted to access PKILabServer.com and firefox blocked my attempt asking me to add an exception. After importing the ca.crt certificate, I had no such issue and the page loaded properly. The page that loaded appears to be all the different ciphers supported by the s_server binary.

1. I modified a single byte in server.pem and restarted the server. The only thing that changed was the master key. Everything else more or less stayed the same. The pages are saved as server_original.txt and server_modified.txt with a difference report stored in difference.txt
2. By going to the URL https://localhost:4433, Firefox no longer recognizes the certifcate for that domain. After adding the exception, the page looks no different.

4.4 Task 4: Establishing a TLS/SSL connection with server

For task 4, I did my best to look at the given cpp files and see where the verifying checks are being made. I replaced all the files with what I generated in task two and continued with task 4. I also removed all the client checks in serv.cpp. In serv.cpp is commented where the corresponding checks are to verify the server certificate. It appears as the serv.cpp file is responsible for the key exchange as that's the one with the private key. I'm not sure entirely about this answer, but based on what I can see, the check occurs in line 80 when the private key is checked. Upon this, the key exchange renders true.

4.5 Task 5: Performance Comparison: RSA versus AES

Writing a script named test_enc_dec.py, encrypting message.txt and decrypting message_enc.txt 5000 times using 1024-bit RSA encryption and 128-bit AES. The results showed similar to the openssl speed tests of rsa and aes. It appears that by increasing the size of the key, RSA encryption/decryption slows down significantly more than AES encryption. For RSA, 512 bits goes from 48,875 verifies per second to 1,036 verifies per second. For 1024 bit encryption, RSA tackles 717 signs per second vs 1398 signs per second with AES. The table is below.

	sign	verify	sign/s	verify/s
rsa	512 bits	0.000249s	0.000020s	4021.0	48875.4
rsa	1024 bits	0.001395s	0.000067s	717.1	15035.2
rsa	2048 bits	0.009388s	0.000253s	106.5	3956.4

rsa 4096 bits 0.066423s 0.000965s 15.1 1036.1

The 'numbers' are in 1000s of bytes per second processed.

type	16 bytes	64 bytes	256 bytes	1024 bytes	8192 bytes
aes-128 cbc	60175.48k	67157.32k	69192.43k	139816.34k	140931.54k
aes-192 cbc	50966.40k	55251.75k	57742.73k	118236.77k	120680.17k
aes-256 cbc	44533.25k	48390.73k	49036.34k	102599.94k	103615.80k

4.6 Task 6: Create Digital Signature

The commands used for this part are the following:

1. Generating an RSA public/private key pair: openssl genrsa -aes128 -out server.key 1024
2. Signing the SHA246 hash of example.txt into example.sha256:

   openssl dgst -sha256 example.txt > example.sha256

   openssl rsautl -sign -inkey server.key -keyform PEM -in example.sha256 > example.signature.sha256
3. verifying the digital signature in example.sha256 :

   openssl rsautl -verify -inkey server.key -keyform PEM -in example.signature.sha256
4. Modifing example.txt and verifying digital signature again. I changed with bless and then repeated the steps.

My observations is that doing everything as it's supposed to leads to a successful verification of example.txt but when modifying example.txt and verifying leads to a failure in verification of the digital signature.