Alex Huddleston
CSCE 465 - 500

Lab Report 5

Paper-and-Pencil Problems

5.3

In *Introduction* we discuss the devious secretary Bob having an automatic means of generating many messages that Alice would sign, and many messages that Bob would like to send. By the birthday problem, by the time Bob has tried a total of 2^{32} messages, he will probably have found two with the same message digest. The problem is, both may be of the same type, which would not do him any good. How many messages must Bob try before it is probable that he'll have messages with matching digests, and that the messages will be of opposite types?

Since the first issue was to find any pair of messages that had the same digest, we concluded we would need to generate 2^{32} messages. If we then want to find 2 messages with the same digest, but different types, similar logic can be applied. We know by former proof that 2^{32} messages will have two messages with the same digest, therefore we need to try 2^{32} combinations of pairs of messages split into 32 pairs of wordings in order to prove that we will have two messages with the same digest and different types. So, that means we need to do 2^{32} * 2^{32} = 2^{64} message combinations.

5.4

In 5.2.4.2 *Hasing Large Messages*, we described a hash algorithm in which a constant was successively encrypted with blocks of the message. We showed that you could find two messages with the same hash value in about 2^{32} operations. So we suggested doubling the hash size by using the message twice, first in forward order to make up the first half of the hash, and then in reverse order for the second half of the hash. Assuming a 64-bit encryption block, how could you find two messages with the same hash value in about 2^{32} iterations? Hint: consider blockwise palindromic messages.

If we only need 2^{32} operations to find a hash collision with the original formula, then we can probably do 2^{32} operations to find a collision with the second. Since the second formula simply uses the same message, but reversed and appended to the end of it, we still only need to try 2^{32} possible messages (since the problem space is still going to contain 2^{32} messages), but when we try to find collisions, we simply do what the second formula asks us to and append the reverse of the message to the original message before feeding it to find the digest.

5.14

For purposes of this exercise, we will define **random** as having all elements equally likely to be chosen. So a function that selects a 100-bit number will be random if every 100-bit number is equally likely to be chosen. Using this definition, if we look at the function "+" and we have two inputs, x and y, then the output will be random if at least one of x and y are random. For instance, y can always be 51, and yet the output will be random if x is random. For the following functions, find sufficient conditions for x, y, and z under which the output will be random:

~X	X is not random
X (+) Y	X or Y are random, but not both of them.
X V Y	X or Y are random.
X /\ Y	Both X and Y are random
(X /\ Y)V(~X /\ Z) [the selection function]	If X is random, then it's random if Y is random. If X is not random, then it's random if Z is random.
(X /\ Y)V(X /\ Z)V (Y /\ Z) [the majority function]	It's random if any two of X, Y, and Z are.
X(+)Y(+)Z	If at least one of X, Y, or Z are random, but not all three of them.
Y(+)(XV-Z)	(assuming "-" is a negative sign, not a "not" sign) If Y is true and both X and Z are false or if Y is false and either X or Z or both are true.

6.2

In section 6.4.2 *Defenses Against Man-in-the-Middle Attack*, it states that encrypting the Diffie-Hellman value with the other side's public key prevents the attack. Why is this the case, given that an attacker can encrypt whatever it wants with the other side's public key?

The attacker both needs to know both sides public keys and needs to know what the Diffie-Hellman value calculated is. The man in the middle won't be able to decrypt what the value was and will fail when trying to send an attack message over.

6.8

Suppose Fred sees your RSA signature on m_1 and on m_2 (i.e. he sees m_1^d mod n and m_2^d mod n). How does he compute the signature on each of m_1^j mod n (for positive integer j), m_1^{-1} mod n, m_1 dot m_2 mod n, and in general m_1^j dot m_2^k mod n (for arbitrary integers j and k)?

I don't understand how to answer this other than saying he needs to find m_1, m_2, d, and n.

Task 1: Generating Message Digest and MAC

In this task, we will play with various one-way hash algorithms. You can use the following openssl dgst command to generate the hash value for a file. To see the manuals, you can type man openssl and man dgst.

% openssl dgst dgsttype filename

Please replace the dgsttype with a specific one-way hash algorithm, such as -md5, -sha1, -sha256, etc. In this task, you should try at least 3 different algorithms, and describe your observations. You can find the supported one-way hash algorithms by typing “man openssl”.

```
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 gpl-3.0.txt
MD5(gpl-3.0.txt)= 1ebbd3e34237af26da5dc08a4e440464
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sh1 gpl-3.0.txt
dgst: Unknown digest sh1
dgst: Use -help for summary.
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha1 gpl-3.0.txt
SHA1(gpl-3.0.txt)= 31a3d460bb3c7d98845187c716a30db81c44b615
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md4 gpl-3.0.txt
MD4(gpl-3.0.txt)= 7cec43f5d53168ea749fa42a15b90142
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha256 gpl-3.0.txt
SHA256(gpl-3.0.txt)= 3972dc9744f6499f0f9b2dbf76696f2ae7ad8af9b23dde66d6af86c9dfb36986
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha512 gpl-3.0.txt
SHA512(gpl-3.0.txt)= d361e5e8201481c6346ee6a886592c51265112be550d5224f1a7a6e116255c2f1ab8788df579d9b8372ed7bfd19bac4b6e70e00b472642966ab5b31
9b99a2686
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$
```

(plaintext of the screenshot)

[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 gpl-3.0.txt
MD5(gpl-3.0.txt)= 1ebbd3e34237af26da5dc08a4e440464
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sh1 gpl-3.0.txt
dgst: Unknown digest sh1
dgst: Use -help for summary.
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha1 gpl-3.0.txt
SHA1(gpl-3.0.txt)= 31a3d460bb3c7d98845187c716a30db81c44b615
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md4 gpl-3.0.txt
MD4(gpl-3.0.txt)= 7cec43f5d53168ea749fa42a15b90142
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha256 gpl-3.0.txt
SHA256(gpl-3.0.txt)=
3972dc9744f6499f0f9b2dbf76696f2ae7ad8af9b23dde66d6af86c9dfb36986
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha512 gpl-3.0.txt
SHA512(gpl-3.0.txt)=
d361e5e8201481c6346ee6a886592c51265112be550d5224f1a7a6e116255c2f1ab8788df579d9b8372ed7bfd19bac4b6e70e00b472642966ab5b319b99a2686

SHA seems to have a fixed length depending on the bit size being used. I tried two files, one with the gpl 3.0 license and one with just a simple sentence, and got the same lengths. Additionally, when trying the same text multiple times, I receive the same output.

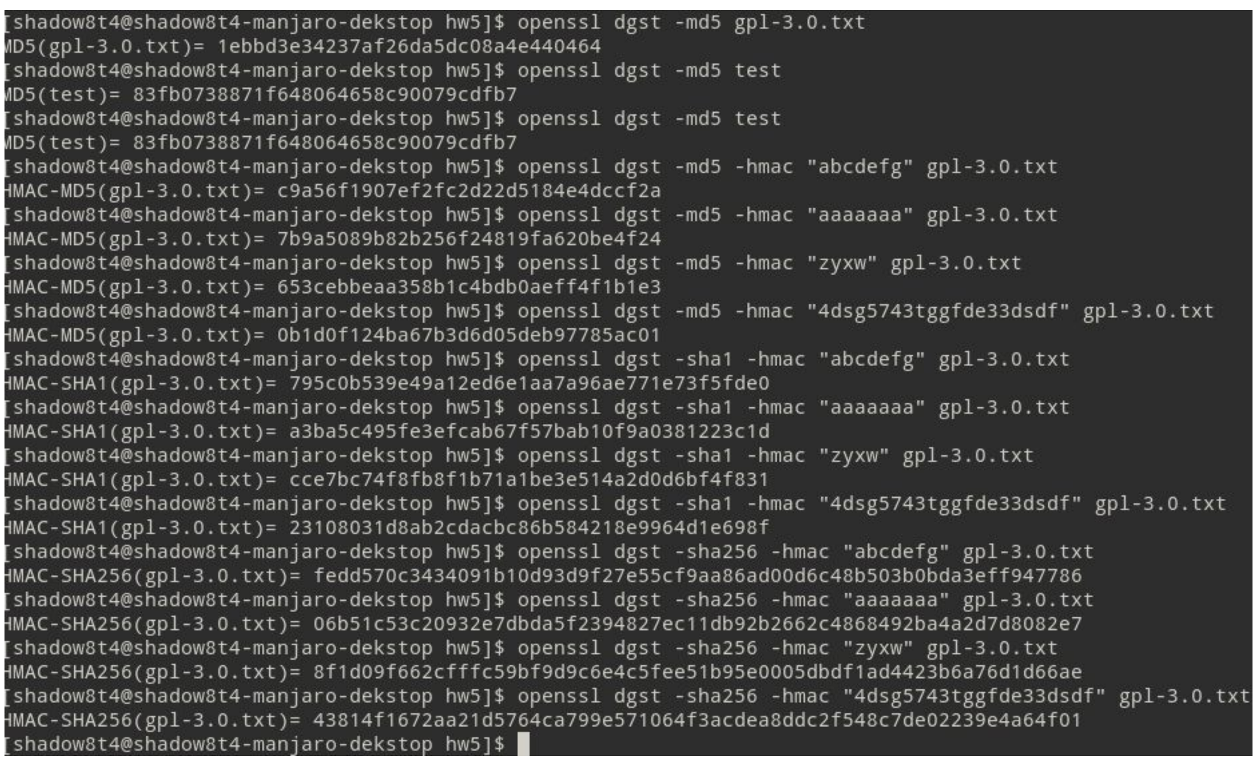
MD seems to have similar properties. I tried MD4 and 5 with the two files, received exactly the same length outputs and when the command was repeated I got the same output.

Task 2: Keyed Hash and HMAC

In this task, we would like to generate a keyed hash (i.e. MAC) for a file. We can use the -hmac option (this option is currently undocumented, but it is supported by openssl). The following example generates a keyed hash for a file using the HMAC-MD5 algorithm. The string following the -hmac option is the key.

% openssl dgst -md5 -hmac “abcdefg” filename

Please generate a keyed hash using HMAC-MD5, HMAC-SHA256, and HMAC-SHA1 for any file that you choose. Please try several keys with different length. Do we have to use a key with a fixed size in HMAC? If so, what is the key size? If not, why?



```
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 gpl-3.0.txt
MD5(gpl-3.0.txt)= 1ebbd3e34237af26da5dc08a4e440464
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 test
MD5(test)= 83fb0738871f648064658c90079cdfb7
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 test
MD5(test)= 83fb0738871f648064658c90079cdfb7
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 -hmac "abcdefg" gpl-3.0.txt
HMAC-MD5(gpl-3.0.txt)= c9a56f1907ef2fc2d22d5184e4dccf2a
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 -hmac "aaaaaaa" gpl-3.0.txt
HMAC-MD5(gpl-3.0.txt)= 7b9a5089b82b256f24819fa620be4f24
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 -hmac "zyxw" gpl-3.0.txt
HMAC-MD5(gpl-3.0.txt)= 653cebbeaa358b1c4bdb0aeff4f1b1e3
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 -hmac "4dsg5743tggfde33dsdf" gpl-3.0.txt
HMAC-MD5(gpl-3.0.txt)= 0b1d0f124ba67b3d6d05deb97785ac01
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha1 -hmac "abcdefg" gpl-3.0.txt
HMAC-SHA1(gpl-3.0.txt)= 795c0b539e49a12ed6e1aa7a96ae771e73f5fde0
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha1 -hmac "aaaaaaa" gpl-3.0.txt
HMAC-SHA1(gpl-3.0.txt)= a3ba5c495fe3efcab67f57bab10f9a0381223c1d
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha1 -hmac "zyxw" gpl-3.0.txt
HMAC-SHA1(gpl-3.0.txt)= cce7bc74f8fb8f1b71a1be3e514a2d0d6bf4f831
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha1 -hmac "4dsg5743tggfde33dsdf" gpl-3.0.txt
HMAC-SHA1(gpl-3.0.txt)= 23108031d8ab2cdacbc86b584218e9964d1e698f
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha256 -hmac "abcdefg" gpl-3.0.txt
HMAC-SHA256(gpl-3.0.txt)= fedd570c3434091b10d93d9f27e55cf9aa86ad00d6c48b503b0bda3eff947786
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha256 -hmac "aaaaaaa" gpl-3.0.txt
HMAC-SHA256(gpl-3.0.txt)= 06b51c53c20932e7dbda5f2394827ec11db92b2662c4868492ba4a2d7d8082e7
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha256 -hmac "zyxw" gpl-3.0.txt
HMAC-SHA256(gpl-3.0.txt)= 8f1d09f662cfffc59bf9d9c6e4c5fee51b95e0005dbdf1ad4423b6a76d1d66ae
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha256 -hmac "4dsg5743tggfde33dsdf" gpl-3.0.txt
HMAC-SHA256(gpl-3.0.txt)= 43814f1672aa21d5764ca799e571064f3acdea8ddc2f548c7de02239e4a64f01
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$
```

(plaintext of screenshot)

[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 gpl-3.0.txt

MD5(gpl-3.0.txt)= 1ebbd3e34237af26da5dc08a4e440464

[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 test

MD5(test)= 83fb0738871f648064658c90079cdfb7

[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 test

MD5(test)= 83fb0738871f648064658c90079cdfb7
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 -hmac "abcdefg" gpl-3.0.txt
HMAC-MD5(gpl-3.0.txt)= c9a56f1907ef2fc2d22d5184e4dccf2a
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 -hmac "aaaaaaa" gpl-3.0.txt
HMAC-MD5(gpl-3.0.txt)= 7b9a5089b82b256f24819fa620be4f24
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 -hmac "zyxw" gpl-3.0.txt
HMAC-MD5(gpl-3.0.txt)= 653cebbeaa358b1c4bdb0aeff4f1b1e3
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 -hmac "4dsg5743tggfde33dsdf" gpl-3.0.txt
HMAC-MD5(gpl-3.0.txt)= 0b1d0f124ba67b3d6d05deb97785ac01
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha1 -hmac "abcdefg" gpl-3.0.txt
HMAC-SHA1(gpl-3.0.txt)= 795c0b539e49a12ed6e1aa7a96ae771e73f5fde0
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha1 -hmac "aaaaaaa" gpl-3.0.txt
HMAC-SHA1(gpl-3.0.txt)= a3ba5c495fe3efcab67f57bab10f9a0381223c1d
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha1 -hmac "zyxw" gpl-3.0.txt
HMAC-SHA1(gpl-3.0.txt)= cce7bc74f8fb8f1b71a1be3e514a2d0d6bf4f831
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha1 -hmac "4dsg5743tggfde33dsdf" gpl-3.0.txt
HMAC-SHA1(gpl-3.0.txt)= 23108031d8ab2cdacbc86b584218e9964d1e698f
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha256 -hmac "abcdefg" gpl-3.0.txt
HMAC-SHA256(gpl-3.0.txt)= fedd570c3434091b10d93d9f27e55cf9aa86ad00d6c48b503b0bda3eff947786
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha256 -hmac "aaaaaaa" gpl-3.0.txt
HMAC-SHA256(gpl-3.0.txt)= 06b51c53c20932e7dbda5f2394827ec11db92b2662c4868492ba4a2d7d8082e7
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha256 -hmac "zyxw" gpl-3.0.txt
HMAC-SHA256(gpl-3.0.txt)= 8f1d09f662cfffc59bf9d9c6e4c5fee51b95e0005dbdf1ad4423b6a76d1d66ae
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha256 -hmac "4dsg5743tggfde33dsdf" gpl-3.0.txt
HMAC-SHA256(gpl-3.0.txt)= 43814f1672aa21d5764ca799e571064f3acdea8ddc2f548c7de02239e4a64f01

I was able to successfully run all these commands with varying keys at varying lengths on each of the encryption schemes. From what I can tell, HMAC does not **require** a length for the key, but it is generally recommended to keep the key random and at a 128-bit length.

Task 3: The Randomness of One-way Hash

To understand the properties of one-way hash functions, we would like to do the following exercise for MD5 and SHA256:

1. Create a text file of any length.
2. Generate the hash value H_1 for this file using a specific hash algorithm.
3. Flip one bit of the input file. You can achieve this modification using ghex.
4. Generate the hash value H_2 for the modified file.
5. Please observe whether H_1 and H_2 are similar or not. Please describe your observations in the report. You can write a short program to count how many bits are the same between H_1 and H_2.

First, I created two files (test and test-flipped) to be my inputs.

```
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ ls
gpl-3.0.txt  test  test-flipped
```

I then computed MD5 and SHA256 hashed for each file. Here were the results:

```
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha256 test
SHA256(test)= 40b0ad0841a9dd691004158e89404be3488f5e7afaf60a6223d86e5db2217340
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -sha256 test-flipped
SHA256(test-flipped)= d311724ee6d7ef1f69d637f2ce19ddcf5a1198efad2085c28ce7d834eb869b76
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 test
MD5(test)= 83fb0738871f648064658c90079cdfb7
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl dgst -md5 test-flipped
MD5(test-flipped)= 8b1c510eb4e3f8cd21bab07416c79e72
```

In plain text, here are the resulting hashes:

<u>MD5</u>

```
Test:           83fb0738871f648064658c90079cdfb7

test-flipped :  8b1c510eb4e3f8cd21bab07416c79e72
```

There are 62 similar bits out of the 128 of each hash between these two hashes.

<u>SHA256</u>

```
Test:
40b0ad0841a9dd691004158e89404be3488f5e7afaf60a6223d86e5db2217340

Test-flipped:
d311724ee6d7ef1f69d637f2ce19ddcf5a1198efad2085c28ce7d834eb869b76
```

There are 119 similar bits out of the 255 of each hash between these two hashes.

We flipped one bit from the text and in return received at least half of the bits being changed from the previous iteration.

Task 4: Hash Collision-Free Property

In this task, we will investigate hash function's collision-free properties. We will use the brute-force method to see how long it takes to break these properties. Instead of using openssl's command-line tools, you are required to write your own C programs to invoke the message digest functions in openssl's crypto library. A sample code can be found from http://www.openssl.org/docs/crypto/EVP_DigestInit.html . Please get familiar with this sample code.

Since most of the hash functions are quite strong against the brute-force attack on those two properties, it will take us years to break them using the brute-force method. To make the task feasible, we reduce the length of the hash value to 24 bits. We can use any one-way hash function, but we only use the first 24 bits of the hash value in this task. Namely, we are using a modified one-way hash function. Please design an experiment to find out the following:

1. How many trials it will take you to find two messages with the same hash values using the brute-force method? You should repeat your experiment for multiple times, and report your average number of trials.
2. How many trials will it take you to find a message that has the same hash value as a given/known message's hash value using the brute-force method? Similarity, you should report the average.
3. Based on your observation, which case is easier to break using the brute-force method?
4. (10 Bonus Points) Can you explain the difference in your observation mathematically (i.e., a formal proof)?

I ran out of time to do this task.

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 a 1024-bit RSA public/private key pair. Then, do the following:

1. Encrypt message.txt using the public key; save 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 spend on each of the above operations, and describe your observations. If an operation is too fast, you may want to repeat it 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

Benchmarks:

```
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl speed rsa
Doing 512 bit private rsa's for 10s: 235363 512 bit private RSA's in 10.00s
Doing 512 bit public rsa's for 10s: 3646937 512 bit public RSA's in 10.00s
Doing 1024 bit private rsa's for 10s: 84224 1024 bit private RSA's in 10.00s
Doing 1024 bit public rsa's for 10s: 1436361 1024 bit public RSA's in 10.00s
Doing 2048 bit private rsa's for 10s: ^C
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl speed aes
Doing aes-128 cbc for 3s on 16 size blocks: 29037753 aes-128 cbc's in 2.99s
Doing aes-128 cbc for 3s on 64 size blocks: 7802452 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 256 size blocks: 2041742 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 1024 size blocks: 1015245 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 8192 size blocks: 129409 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 16384 size blocks: 64486 aes-128 cbc's in 3.00s
Doing aes-192 cbc for 3s on 16 size blocks: 24833471 aes-192 cbc's in 2.99s
Doing aes-192 cbc for 3s on 64 size blocks: 6557674 aes-192 cbc's in 3.00s
Doing aes-192 cbc for 3s on 256 size blocks: ^C
```

(plaintext of screenshot)
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl speed rsa
Doing 512 bit private rsa's for 10s: 235363 512 bit private RSA's in 10.00s
Doing 512 bit public rsa's for 10s: 3646937 512 bit public RSA's in 10.00s
Doing 1024 bit private rsa's for 10s: 84224 1024 bit private RSA's in 10.00s
Doing 1024 bit public rsa's for 10s: 1436361 1024 bit public RSA's in 10.00s
Doing 2048 bit private rsa's for 10s: ^C
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl speed aes
Doing aes-128 cbc for 3s on 16 size blocks: 29037753 aes-128 cbc's in 2.99s
Doing aes-128 cbc for 3s on 64 size blocks: 7802452 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 256 size blocks: 2041742 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 1024 size blocks: 1015245 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 8192 size blocks: 129409 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 16384 size blocks: 64486 aes-128 cbc's in 3.00s
Doing aes-192 cbc for 3s on 16 size blocks: 24833471 aes-192 cbc's in 2.99s
Doing aes-192 cbc for 3s on 64 size blocks: 6557674 aes-192 cbc's in 3.00s

Encrypting and decrypting by public/private key was negligible time-wise, especially considering the bechmark's results.

Similar results in the aes-128-sbs test for this file, as shown by the benchmark stats listed above.

```
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl rsautl -encrypt -inkey public.pem -pubin -in message.txt -out message.enc.txt
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl rsautl -decrypt -inkey private.pem -in message.enc.txt -out output.txt
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl rsautl -encrypt -inkey public.pem -pubin -in message.txt -out message.enc.txt
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl rsautl -decrypt -inkey private.pem -in message.enc.txt -out output.txt
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl rsautl -encrypt -inkey public.pem -pubin -in message.txt -out message.enc.txt
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl rsautl -decrypt -inkey private.pem -in message.enc.txt -out output.txt
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl rsautl -encrypt -inkey public.pem -pubin -in message.txt -out message.enc.txt
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl rsautl -decrypt -inkey private.pem -in message.enc.txt -out output.txt
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl rsautl -encrypt -inkey public.pem -pubin -in message.txt -out message.enc.txt
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl rsautl -decrypt -inkey private.pem -in message.enc.txt -out output.txt
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl rsautl -encrypt -inkey public.pem -pubin -in message.txt -out message.enc.txt
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl enc -aes-128 -in message.txt -out aes-output.txt
enc: Unknown cipher aes-128
enc: Use -help for summary.
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ openssl enc -aes-128-cbc -in message.txt -out aes-output.txt
enter aes-128-cbc encryption password:
Verifying - enter aes-128-cbc encryption password:
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$ ls
aes-output.txt  gpl-3.0.txt  message.enc.txt  message.txt  output.txt  private.pem  public.pem  test  test-flipped
[shadow8t4@shadow8t4-manjaro-dekstop hw5]$
```

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.

1. Compute the sha256 hash of the exmaple file and append it to example.txt, save as example.sha256. This was done with the following command:
    a. `openssl dgst -sha256 -sign "private.pem" -out example.sha256 example.txt`
2. Verify the digital signature with the following openssl command:
    a. `openssl dgst -sha256 -verify  "public.pem" -signature example.sha256 example.txt`

```
[shadow8t4@shadow8t4-manjaro-dekstop digital_signature]$ openssl dgst -sha256 -sign "private.pem" -out example.sha256 example.txt
[shadow8t4@shadow8t4-manjaro-dekstop digital_signature]$ openssl dgst -sha256 -verify  "public.pem" -signature example.sha256 example.txt
Verified OK
```

3. I modified the file by changing some of the text in example.txt, then went through the same verification process.
    a. If we were meant to modify example.sha256 (as I assume this step was **supposed** to point out the authenticity of verification, and this would be the way to it), then I did that as well.
    b. If I was meant to modify the example.txt and check verification without signing, I also tried that.

```
[shadow8t4@shadow8t4-manjaro-dekstop digital_signature]$ vim example.txt
[shadow8t4@shadow8t4-manjaro-dekstop digital_signature]$ openssl dgst -sha256 -verify  "public.pem" -signature example.sha256 example.txt
Verification Failure
[shadow8t4@shadow8t4-manjaro-dekstop digital_signature]$ openssl dgst -sha256 -verify  "public.pem" -signature example.sha256 example.txt
Verification Failure
[shadow8t4@shadow8t4-manjaro-dekstop digital_signature]$ openssl dgst -sha256 -sign "private.pem" -out example.sha256 example.txt
[shadow8t4@shadow8t4-manjaro-dekstop digital_signature]$ openssl dgst -sha256 -verify  "public.pem" -signature example.sha256 example.txt
Verified OK
[shadow8t4@shadow8t4-manjaro-dekstop digital_signature]$ ghex example.sha256

(ghex:12758): Gtk-WARNING **: Allocating size to GtkDrawingArea 0x1d9b450 without calling gtk_widget_get_preferred_width/height(). How does
the code know the size to allocate?

(ghex:12758): Gtk-WARNING **: Allocating size to GtkDrawingArea 0x1d9b210 without calling gtk_widget_get_preferred_width/height(). How does
the code know the size to allocate?

(ghex:12758): Gtk-WARNING **: Allocating size to GtkDrawingArea 0x1d9b330 without calling gtk_widget_get_preferred_width/height(). How does
the code know the size to allocate?
[shadow8t4@shadow8t4-manjaro-dekstop digital_signature]$ openssl dgst -sha256 -verify  "public.pem" -signature example.sha256 example.txt
Verified OK
[shadow8t4@shadow8t4-manjaro-dekstop digital_signature]$ openssl dgst -sha256 -verify  "public.pem" -signature example-modified.sha256.sha25
6 example.txt
dgst: Cannot open input file example-modified.sha256.sha256, No such file or directory
dgst: Use -help for summary.
[shadow8t4@shadow8t4-manjaro-dekstop digital_signature]$ openssl dgst -sha256 -verify  "public.pem" -signature example-modifiedsha256 exampl
e.txt
dgst: Cannot open input file example-modifiedsha256, No such file or directory
dgst: Use -help for summary.
[shadow8t4@shadow8t4-manjaro-dekstop digital_signature]$ openssl dgst -sha256 -verify  "public.pem" -signature example-modified.sha256 examp
le.txt
Verification Failure
```

References

https://raymii.org/s/tutorials/Encrypt_and_decrypt_files_to_public_keys_via_the_OpenSSL_Command_Line.html
https://raymii.org/s/tutorials/Sign_and_verify_text_files_to_public_keys_via_the_OpenSSL_Command_Line.html
Not sure if I actually copied too much fo those, but I did use them as references.