

Homework 5

Paper and Pencil problem:

5.3/ In §5.1 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?

According to the description, Bob is generating 2^{32} messages of each type and looking for a match between the two lists. The probability that a given Type II message has the same message digest as one the 2^{32} Type I messages is about $2^{32}/2^{64} = 1/2^{32}$, so it is likely that one of the 2^{32} Type II messages matches one the 2^{32} Type I messages.

5.4/ In §5.2.4.2 Hashing 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.

The digest size will remain 64 bits. Therefore, the iterations will remain the same, and if a palindrome exists within the message space, then the message will be the same forward and backward, leading to a collision when the two messages are hashed.

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:

$$\sim x$$

$$x \oplus y$$

$$x \vee y$$

$$x \wedge y$$

$$(x \wedge y) \vee (\sim x \wedge z) \text{ [the selection function]}$$

$$(x \wedge y) \vee (x \wedge z) \vee (y \wedge z) \text{ [the majority function]}$$

$$x \oplus y \oplus z$$

$$y \oplus (x \vee \sim z)$$

- $\sim x$: x is random, y & z are independent
- $x \oplus y$: either x or y or both are random, and they differ in at least one bit position in their binary representation, z is independent.
- $x \vee y$: either x or y or both are random and unequal to each other in at least one bit position in their binary representation, z is independent.
- $x \wedge y$: either x or y or both are random and unequal to each other and both have 1 in at least one bit position in their binary representation, z is independent.
- $(x \wedge y) \vee (\sim x \wedge z)$: either x and y are different so as to obtain a non-zero value in the left-hand side of the function or $\sim x$ and z differ so as to get a unique output for the overall OR function.
- $(x \wedge y) \vee (x \wedge z) \vee (y \wedge z)$: either one of x or y or z is different in at least one bit position in their binary representation.
- $x \oplus y \oplus z$: either two of the three are different in at least one bit position in their binary representation.
- $y \oplus (x \vee \sim z)$: either of x or $\sim z$ is different so as to get the combination on the right-hand side different to y , in at least one bit position in their binary representation.

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 hacker will not be able to decrypt the Diffie Helman values. Because of this, he will not be able to compute the shared secrets.

6.8/ Suppose Fred sees your RSA signature on m_1 and on m_2 , How does he compute the signature on each of the $m_1 \bmod n$, $m_1^{-1} \bmod n$, $m_1 * m_2 \bmod n$, and in general $m_1^j * m_2^k \bmod n$ (for arbitrary integers and k)?

Let's use S_1 and S_2 for the given signatures of m_1 and m_2 .

$$S_1 = m_1^{d/k} \bmod n$$

$$S_2 = m_2^{d/k} \bmod n$$

The signature of $m_1^j = s_1^j \bmod n$, because $(m_1^j)^d \bmod n = (m_1^d)^j \bmod n$. This means that assuming there's a multiplicative inverse we can use the equation:

- $(m_1^{-1}) = s_1^{-1} \bmod n$ assuming m_1^{-1} and m_2^{-1} exist.

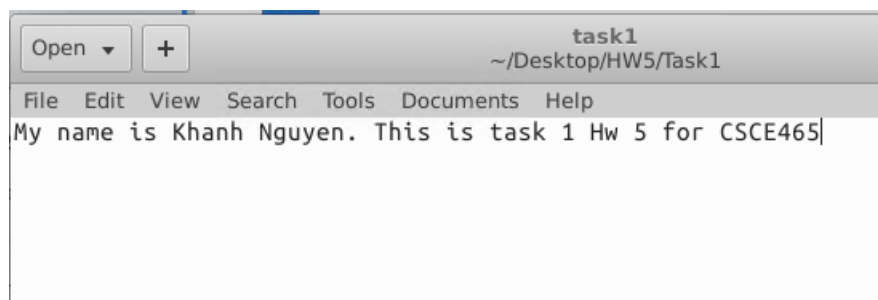
When calculating the signatures $m_1 * m_2$ it would be written as $s_1 * s_2 \bmod n$.

Therefore, in order for Fred to calculate the signature (assuming there j and k are arbitrary integers) $m_1^j * m_2^k$

$$(m_1^j * m_2^k) = s_1^j * s_2^k \bmod n.$$

Task 1: Generating Message Digest and MAC:

For this task, I created a file named "task1.txt" and used `openssl dgst` command to generate hash value for the file. I used three algorithms which are MD5, SHA-1, SHA-256. The result can be seen below:



Content of task1.txt

```
Evaluation-SEED$$ openssl dgst -md5 task1.txt
MD5(task1.txt)= 817e962d591915824748cf7d87ee84ef
[04/12/19]seed@VM:~/.../Task1$
Evaluation-SEED$$ openssl dgst -sha1 task1.txt
SHA1(task1.txt)= 4faa503b8e942c770e0eec6980aa8b781f6f3932
[04/13/19]seed@VM:~/.../Task1$
Evaluation-SEED$$ openssl dgst -sha256 task1.txt
SHA256(task1.txt)= 0dceca2e82ecfe5d5de63f4d4e2f53bf459acd87ecab5878027ed56f4f15c
70d
```

Results of hashing with 3 different techniques

The length of hash value for each technique is different. For MD5, the hash value has 32 characters, in other words, hash value of MD5 is 128 bits. For SHA-1, it resulted in 40 character long, so the hash value of SHA-1 is 160 bits. Similarly, SHA-256 has the longest hash value, which is 256 bits.

Task 2: Keyed Hash and HMAC:

For task 2, I used the same file in task 1. I used different keys ('abc', 'abcd', and 'abcdefgh') to generate hash values for the file using different techniques HMAC-MD5, HMC-SHA256, HMAC-SHA1. The results can be seen below:

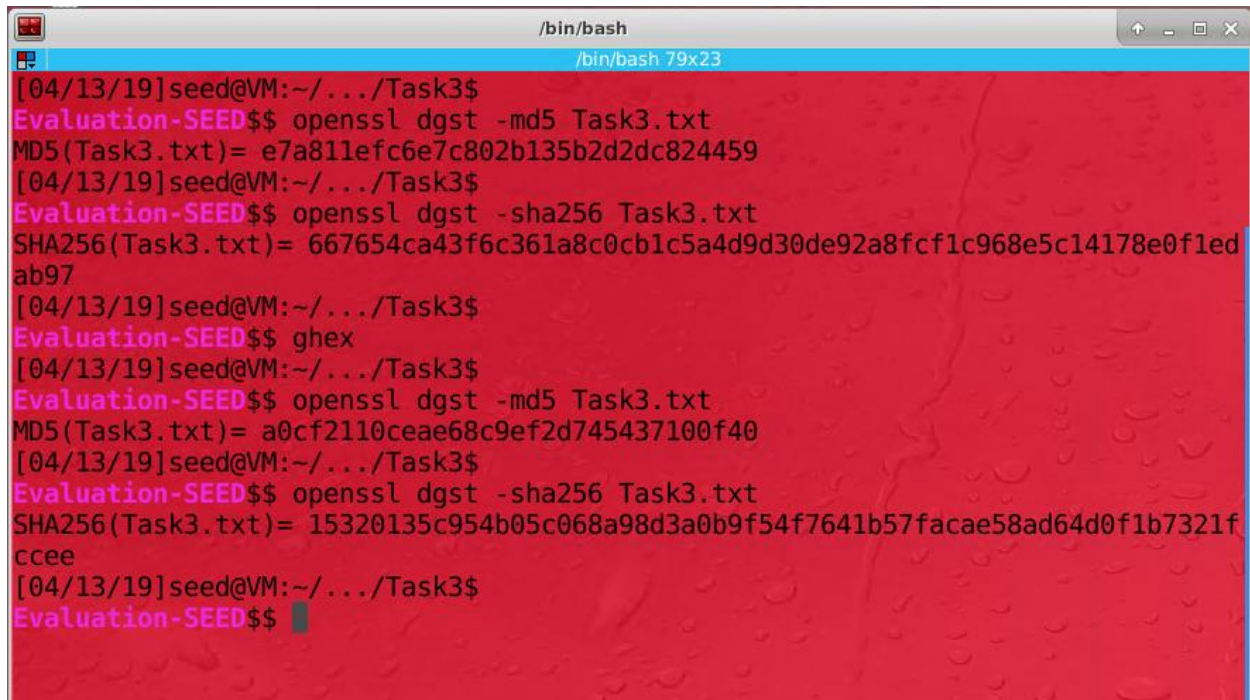
```
[04/13/19]seed@VM:~/.../Task1$  
Evaluation-SEED$ openssl dgst -md5 -hmac "abc" task1.txt  
 HMAC-MD5(task1.txt)= a001dcc395126068c68fafb679542f42  
[04/13/19]seed@VM:~/.../Task1$  
Evaluation-SEED$ openssl dgst -md5 -hmac "abcde" task1.txt  
 HMAC-MD5(task1.txt)= 12df87fe8411f8578b422f611583420c  
[04/13/19]seed@VM:~/.../Task1$  
Evaluation-SEED$ openssl dgst -md5 -hmac "abcdefg" task1.txt  
 HMAC-MD5(task1.txt)= c0bd8ea664602c68374ba7bfd0d9587a  
[04/13/19]seed@VM:~/.../Task1$  
Evaluation-SEED$ openssl dgst -sha256 -hmac "abc" task1.txt  
 HMAC-SHA256(task1.txt)= 778ceb69579f728181140c4eb9159ff2f38725c5618a43991a3cc83dfb4  
57e14  
[04/13/19]seed@VM:~/.../Task1$  
Evaluation-SEED$ openssl dgst -sha256 -hmac "abcde" task1.txt  
 HMAC-SHA256(task1.txt)= 84173527f67008ba3ac9c81c6c04daea34e75a8f6beb81fe55da6c0a674  
f65c3  
[04/13/19]seed@VM:~/.../Task1$  
Evaluation-SEED$ openssl dgst -sha256 -hmac "abcdefg" task1.txt  
 HMAC-SHA256(task1.txt)= 109971f5aaf10943be09deb2c002d7700d60cad961de68490c7038d9622  
549c9  
[04/13/19]seed@VM:~/.../Task1$  
Evaluation-SEED$ openssl dgst -sha1 -hmac "abc" task1.txt  
 HMAC-SHA1(task1.txt)= 0a0cc85d8ae90acc7a7b142623c25f735b948daf  
[04/13/19]seed@VM:~/.../Task1$  
Evaluation-SEED$ openssl dgst -sha1 -hmac "abcde" task1.txt  
 HMAC-SHA1(task1.txt)= e87f56e1aa6488d07da66ab8200ce9efdee9f7d5  
[04/13/19]seed@VM:~/.../Task1$  
Evaluation-SEED$ openssl dgst -sha1 -hmac "abcdefg" task1.txt  
 HMAC-SHA1(task1.txt)= 6fe680b89968a920e35c7bd627e382cef0285f43  
[04/13/19]seed@VM:~/.../Task1$  
Evaluation-SEED$
```

Keyed hash with different key lengths and techniques

From the results, we can see that for each of the techniques, the keyed hash values have the same length no matter how long the key is. Therefore, I don't think we have to use the key with fixed size in HMAC.

Task 3: The Randomness of One-way Hash:

In this task, I created a file task3.txt then generated hash values for it using MD5 and SHA256 techniques. Then I modified one bit of the file using ghex. After that I generated MD5 and SHA256 hash value for the modified file. The results can be observed below:



```
/bin/bash
/bin/bash 79x23
[04/13/19]seed@VM:~/.../Task3$
Evaluation-SEED$ openssl dgst -md5 Task3.txt
MD5(Task3.txt)= e7a811efc6e7c802b135b2d2dc824459
[04/13/19]seed@VM:~/.../Task3$
Evaluation-SEED$ openssl dgst -sha256 Task3.txt
SHA256(Task3.txt)= 667654ca43f6c361a8c0cb1c5a4d9d30de92a8fcf1c968e5c14178e0f1ed
ab97
[04/13/19]seed@VM:~/.../Task3$
Evaluation-SEED$ ghex
[04/13/19]seed@VM:~/.../Task3$
Evaluation-SEED$ openssl dgst -md5 Task3.txt
MD5(Task3.txt)= a0cf2110ceae68c9ef2d745437100f40
[04/13/19]seed@VM:~/.../Task3$
Evaluation-SEED$ openssl dgst -sha256 Task3.txt
SHA256(Task3.txt)= 15320135c954b05c068a98d3a0b9f54f7641b57facae58ad64d0f1b7321f
ccee
[04/13/19]seed@VM:~/.../Task3$
Evaluation-SEED$
```

Task 3 results

From the result, we can see that changing one bit of the file can result in a totally different hash value. There are only 3 bits got repeated in MD5 and 1 bit got repeated in SHA256.

Task 4: Hash Collision-Free Properties:

For this task, I used Python with built-in hashlib library to implement and experiment the two properties of hashing: Collision-Free Property and One-Way Property. The random strings I generated range from 1 to 26 characters. I used MD5 technique to generate hash values for random generated strings.

- For Collision-Free Property: I used a dictionary to store all the generated strings and hash values. Then I checked if the newly generated hash value already existed in the dictionary. If the dictionary already contains the hash value that its first 6 hex bits are the same as different other hash values's, then the Collision-Free Property has been cracked.
- For One-Way Property: I created a fixed string "Khanh Nguyen task 4" and generated its hash value. After that I generated other random strings and compared their hash values with the original one until I found collision in the first 6 hex bits.

The code is shown below:


```
1 import hashlib
2 import string
3 import time
4 import random
5 from random import randint
6
7 def getHash(msg):
8     return hashlib.md5(msg).hexdigest()
9
10 def getRdmStr(len):
11     return ''.join(random.choice(string.ascii_letters + string.punctuation) for i in range(len))
12
13 def one_way_property(originalMsg):
14     originalHash = getHash(originalMsg)
15     originalHash_6bits = originalHash[0:6] # We only compare first 24 bits binary, which equals to 6 bits hex
16     dupHash_6bits = None
17     count = 0
18
19     while (originalHash_6bits != dupHash_6bits):
20         count += 1
21         randomMsg = getRdmStr(randint(1,26)) # We generate random string of length 1-26
22         dupHash = getHash(randomMsg)
23         dupHash_6bits = dupHash[0:6]
24
25     print('Original Message = %-27s\nHash 1 = %32s' % (str(originalMsg), str(originalHash)))
26     print('Duplicate Message = %-27s\nHash 2 = %32s' % (str(randomMsg), str(dupHash)))
27     print('Number of Iterations = %d' % count)
28
29 def collision_free():
30     count = 0
31     hashDict = dict() # dictionary to store all hash values generated
32
33     while (True):
34         count += 1
35         randomMsg = getRdmStr(randint(1,26))
36         hashValue = getHash(randomMsg)
37         hash_6bits = hashValue[0:6]
38
39         if hash_6bits in hashDict and hashDict[hash_6bits] != randomMsg:
40             dupHash = getHash(hashDict[hash_6bits])
41             print('Message 1 = %-27s\nHash 1 = %32s' % (randomMsg, str(hashValue)))
42             print('Message 2 = %-27s\nHash 2 = %32s' % (hashDict[hash_6bits], str(dupHash)))
43             print('Number of Iterations = %d' % count)
44             return count
45         hashDict[hash_6bits] = randomMsg
46
47
48 if __name__ == "__main__":
49     totalNumTrials = 0
50     for i in range(1,6):
51         print('NUMBER OF TRIALS: %d' % i)
52         print('ONE WAY PROPERTY:')
53         one_way_property('Khanh Nguyen task 4')
54
55         # Uncomment this to run collision-free crack and comment out one_way_property
56         # print('COLLISION FREE:')
57         # totalNumTrials += collision_free()
58
59     print('-----')
60     print('Average number of trials: %d' % (totalNumTrials/5))
61
62
```

Task 4 code for Collision Free Property and One Way Property

For each property, I ran cracking code 5 times to find the average number of trials. The result is shown below:

```
/bin/bash
/bin/bash 75x40
Hash 1 = 937963d1ba93834add8be0baf020cf63
Message 2 = a|xi|%=cT)
Hash 2 = 9379632ad3b503ddf5ed362aa96dae7d
Number of Iterations = 3584
-----
NUMBER OF TRIALS: 2
COLLISION FREE:
Message 1 = aCMxjehSz$}
Hash 1 = 8b5877d88e3b90ae4945dc59753b9f5c
Message 2 = *a!'i&u
Hash 2 = 8b58774f46ed2eb7725edab774d3fc5c
Number of Iterations = 4158
-----
NUMBER OF TRIALS: 3
COLLISION FREE:
Message 1 = :dnqlnawjTL/@Ti)v#D;*luR^q
Hash 1 = ad5b8a9aa9eb8438c3c08cd34e3e6ea7
Message 2 = |D)0H^0~>CT|BX{mPFI-.xTH
Hash 2 = ad5b8a6ba72f2d55d1d95216f231cf80
Number of Iterations = 7245
-----
NUMBER OF TRIALS: 4
COLLISION FREE:
Message 1 = _-;`+
Hash 1 = c16fc3ca2311ddb9c9bcd163f6e2aeeb
Message 2 = pz"IJ+D}{ ' EAS@
Hash 2 = c16fc3a30e2b3dfde50cddaf781aa68f
Number of Iterations = 6838
-----
NUMBER OF TRIALS: 5
COLLISION FREE:
Message 1 = u<QA|)t
Hash 1 = 6dd75757c406600231ebc5750e189250
Message 2 = Mb
Hash 2 = 6dd757cbdd852a16f222a7d1a07eab3e
Number of Iterations = 5396
-----
Average number of trials: 5444
[04/19/19]seed@VM:~/.../Task4$
```

Result for breaking collision free property


```
/bin/bash
/bin/bash 75x40
Hash 1 = b889682c987313630d2b7aa407b83c46
Duplicate Message = ^hTN@SwyGQH
Hash 2 = b88968ff9e7e4c1c1f0fbee0f8c6bd02
Number of Iterations = 2089485
-----
NUMBER OF TRIALS: 2
ONE WAY PROPERTY:
Original Message = Khanh Nguyen task 4
Hash 1 = b889682c987313630d2b7aa407b83c46
Duplicate Message = ;#i&d+N"'>jBU&G>UB=}ifr
Hash 2 = b88968572dc9a0a84bd024097c5b4bc6
Number of Iterations = 44154520
-----
NUMBER OF TRIALS: 3
ONE WAY PROPERTY:
Original Message = Khanh Nguyen task 4
Hash 1 = b889682c987313630d2b7aa407b83c46
Duplicate Message = !UFH?
Hash 2 = b88968b7639c5ff25321a4e5d9b5d5ba
Number of Iterations = 17484825
-----
NUMBER OF TRIALS: 4
ONE WAY PROPERTY:
Original Message = Khanh Nguyen task 4
Hash 1 = b889682c987313630d2b7aa407b83c46
Duplicate Message = Z*\lQ"Bmd|u(Y]&:UY/%r
Hash 2 = b889682f6bdc3d9ece3274f6b0c2b8a4
Number of Iterations = 6261519
-----
NUMBER OF TRIALS: 5
ONE WAY PROPERTY:
Original Message = Khanh Nguyen task 4
Hash 1 = b889682c987313630d2b7aa407b83c46
Duplicate Message = p:D=GFbo'#{#y&e
Hash 2 = b88968c5d2c73501ab5937116e58be7a
Number of Iterations = 13799848
-----
Average number of trials: 16758039
[04/19/19]seed@VM:~/.../Task4$
```

Result for breaking One-Way Property

- 1) From the screenshots above, I ran the code 5 times to calculate the average number of trials for Collision Free Property. The average number of trials for breaking Collision Free Property (where I had to find two messages that have same hash values) is 5,444.
- 2) From the screenshots above, I ran the code 5 times to calculate the average number of trials for One Way Property. The average number of trials for breaking One Way Property (where I had to find messages that have same hash values as known message's

hash value) is 16,758,039 which is much larger than the result from breaking Collision Free Property.

- 3) Based on my observation, the One Way Property is much harder to break and took me much longer time to execute the code (much more number of iterations). Collision Free Property is much easier to break and the code ran much faster.
- 4) For the One Way Property, we are finding the matching hash value that are from 16^6 hex bits (6 because we're only comparing the first 6 hex bits or in other words, 24 binary bits). The probability therefore is $1/16,777,216$ which is very closed to the number that I got from my experiment above.

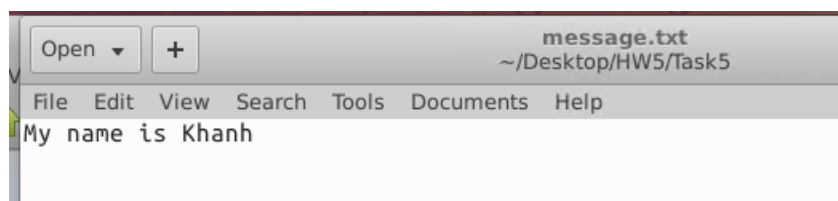
For the Collision Free Property, we are finding the two matching hash values from $16^6 = 16,777,216$ combinations. Each time we generate a hash, we add it to a dictionary and keep comparing them with the new ones. Therefore, the probability for finding matching hash values is much higher.

Considering the scenario of birthday attack, we are having a large number of inputs – 2^{24} , to have the at least 0.5 probability of hash collision, we can hash at least $2^{24/2} = 4096$ messages. Therefore, it can be said that the probability to break to collision free property is approximately $1/4096$.

Task 5: Performance Comparison: RSA versus AES:

These below are the steps for encrypting and decrypting message using 1024 bit RSA:

Before applying the encryption, I created a file 'message.txt' that contain 16-byte text like below:



To encrypt the file using RSA method, first I need to generate the 1024 bit RSA private key:

```
Evaluation-SEED$ openssl genrsa -des3 -out private.pem 1024
Generating RSA private key, 1024 bit long modulus
..+++++
.....+++++
e is 65537 (0x10001)
Enter pass phrase for private.pem:
Verifying - Enter pass phrase for private.pem:
[04/14/19]seed@VM:~/.../Task5$
```

Khanh Nguyen
525000335
CSCE465

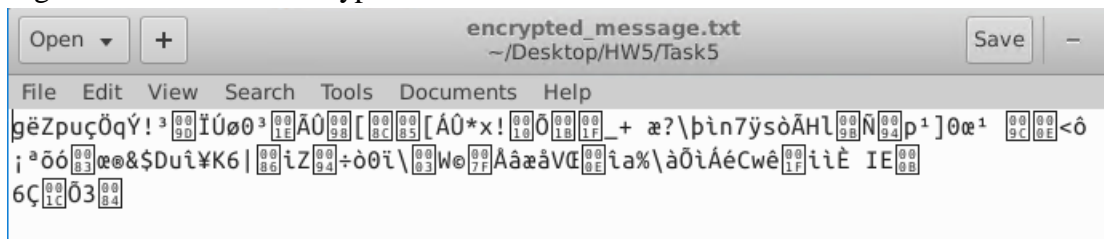
Next, I generated the public key associated the with private key:

```
Evaluation-SEED$$ openssl rsa -in private.pem -out public.pem -outform PEM -pubout
Enter pass phrase for private.pem:
writing RSA key
[04/14/19]seed@VM:~/.../Task5$
```

After that, I encrypt the file with the public key created:

```
Evaluation-SEED$$ openssl rsautl -encrypt -inkey public.pem -pubin -in /home/seed/Desktop/HW5/Task5/message.txt -out /home/seed/Desktop/HW5/Task5/encrypted_message.txt
[04/14/19]seed@VM:~/.../Task5$
```

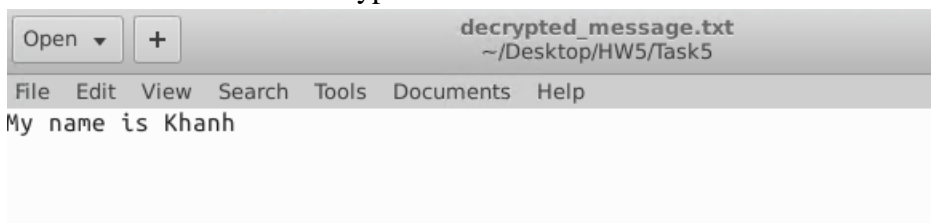
It generated the RSA encrypted file with the content shown below:



Now, I decrypted the file with the private key generated previously:

```
Evaluation-SEED$$ openssl rsautl -decrypt -inkey public.pem -pubin -in /home/seed/Desktop/HW5/Task5/encrypted_message.txt -out /home/seed/Desktop/HW5/Task5/message.txt
[04/14/19]seed@VM:~/.../Task5$
```

And the content of the decrypted file is shown below:



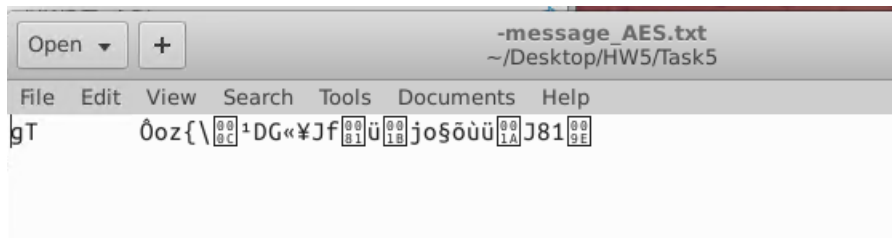
These steps below are for encrypting file using 128-bit AES key:

For this part, I used the same file message.txt. I encrypted the file using 128-bit AES CBC:

```
Evaluation-SEED$$ openssl enc -aes-128-cbc -e -in message.txt -out -message_AES.txt -K 0123456789ABCDEF -iv 0123456789
[04/14/19]seed@VM:~/.../Task5$
```

Khanh Nguyen
525000335
CSCE465

It resulted in the file below:



Since the encrypting for both AES and RSA is pretty fast, I wrote a shell code to run each of the command 1000 times. After that I calculate the average runtime for each:

```
SECONDS=0
value=0

for i in {1..1000}
do
    value=$(( $value + 1))
    duration=$SECONDS
    openssl rsautl -encrypt -inkey public.pem -pubin -in message.txt -out enc_message.txt
    #openssl enc -aes-128-cbc -e -in message.txt -out encrypted_AES_message.txt -K 0123456789ABCDEF -iv 0123456789
done

echo "The program ran $value times."
```

Shell code to run RSA and AES encryption

```
Evaluation-SEED$ time ./test.sh
The program ran 1000 times.

real    0m3.679s
user    0m0.052s
sys     0m0.220s
[04/19/19]seed@VM:~/.../Task5$
```

Result for RSA ran with shell code

```
Evaluation-SEED$ time ./test.sh
The program ran 1000 times.

real    0m3.440s
user    0m0.088s
sys     0m0.192s
```

Result for AES ran with shell code

From the result above, we can see that their runtime are pretty similar. RSA took averagely 3.679 milliseconds for each operation and AES took averagely 3.440 milliseconds for each operation (I ran 1000 times each method). There is no significant difference between the two.

After that, I ran the benchmarking for RSA and AES:

```
Evaluation-SEED$ openssl speed rsa
Doing 512 bit private rsa's for 10s: 60399 512 bit private RSA's in 10.00s
Doing 512 bit public rsa's for 10s: 743104 512 bit public RSA's in 9.99s
Doing 1024 bit private rsa's for 10s: 10836 1024 bit private RSA's in 9.98s
Doing 1024 bit public rsa's for 10s: 227493 1024 bit public RSA's in 10.00s
Doing 2048 bit private rsa's for 10s: 1628 2048 bit private RSA's in 10.01s
Doing 2048 bit public rsa's for 10s: 59040 2048 bit public RSA's in 10.00s
Doing 4096 bit private rsa's for 10s: 220 4096 bit private RSA's in 10.01s
Doing 4096 bit public rsa's for 10s: 9687 4096 bit public RSA's in 10.01s
OpenSSL 1.0.2g 1 Mar 2016
built on: reproducible build, date unspecified
options:bn(64,32) rc4(8x,mmx) des(ptr,risc1,16,long) aes(partial) blowfish(idx)
compiler: cc -I. -I.. -I../include -fPIC -DOPENSSL_PIC -DOPENSSL_THREADS -D_REE
NTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DL_ENDIAN -g -O2 -fstack-protector-strong -Wf
ormat -Werror=format-security -Wdate-time -D_FORTIFY_SOURCE=2 -Wl,-Bsymbolic-fun
ctions -Wl,-z,relro -Wa,--noexecstack -Wall -DOPENSSL_BN_ASM_PART_WORDS -DOPENSS
L_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM
-DSHA512_ASM -DMD5_ASM -DRMD160_ASM -DAES_ASM -DVP_AES_ASM -DWHIRLPOOL_ASM -DGHAS
H_ASM

          sign      verify      sign/s verify/s
rsa 512 bits 0.000166s 0.000013s 6039.9 74384.8
rsa 1024 bits 0.000921s 0.000044s 1085.8 22749.3
rsa 2048 bits 0.006149s 0.000169s 162.6 5904.0
rsa 4096 bits 0.045500s 0.001033s 22.0 967.7
[04/14/19]seed@VM:~/.../Task5$
```

Benchmarking for RSA

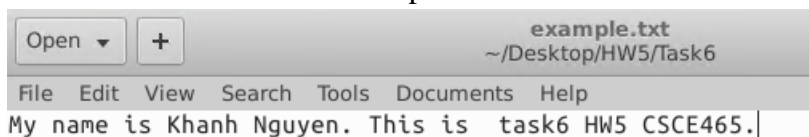

```
Evaluation-SEED$$ openssl speed aes
Doing aes-128 cbc for 3s on 16 size blocks: 15568224 aes-128 cbc's in 2.98s
Doing aes-128 cbc for 3s on 64 size blocks: 4421615 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 256 size blocks: 1141529 aes-128 cbc's in 3.00s
Doing aes-128 cbc for 3s on 1024 size blocks: 288538 aes-128 cbc's in 2.99s
Doing aes-128 cbc for 3s on 8192 size blocks: 36120 aes-128 cbc's in 3.00s
Doing aes-192 cbc for 3s on 16 size blocks: 13355489 aes-192 cbc's in 3.00s
Doing aes-192 cbc for 3s on 64 size blocks: 3719115 aes-192 cbc's in 3.00s
Doing aes-192 cbc for 3s on 256 size blocks: 957568 aes-192 cbc's in 3.00s
Doing aes-192 cbc for 3s on 1024 size blocks: 240141 aes-192 cbc's in 3.00s
Doing aes-192 cbc for 3s on 8192 size blocks: 30134 aes-192 cbc's in 2.99s
Doing aes-256 cbc for 3s on 16 size blocks: 11646628 aes-256 cbc's in 3.00s
Doing aes-256 cbc for 3s on 64 size blocks: 3195495 aes-256 cbc's in 3.00s
Doing aes-256 cbc for 3s on 256 size blocks: 816928 aes-256 cbc's in 3.00s
Doing aes-256 cbc for 3s on 1024 size blocks: 205501 aes-256 cbc's in 3.00s
Doing aes-256 cbc for 3s on 8192 size blocks: 25714 aes-256 cbc's in 3.00s
OpenSSL 1.0.2g 1 Mar 2016
built on: reproducible build, date unspecified
options:bn(64,32) rc4(8x,mmx) des(ptr,risc1,16,long) aes(partial) blowfish(idx)
compiler: cc -I. -I.. -I../include -fPIC -DOPENSSL_PIC -DOPENSSL_THREADS -D_REENTRANT -DDSO_DLFCN -DHAVE_DLFCN_H -DL_ENDIAN -g -O2 -fstack-protector-strong -Wformat -Werror=format-security -Wdate-time -D_FORTIFY_SOURCE=2 -Wl,-Bsymbolic-functions -Wl,-z,relro -Wa,--noexecstack -Wall -DOPENSSL_BN_ASM_PART_WORDS -DOPENSSL_IA32_SSE2 -DOPENSSL_BN_ASM_MONT -DOPENSSL_BN_ASM_GF2m -DSHA1_ASM -DSHA256_ASM -DSHA512_ASM -DMD5_ASM -DRMD160_ASM -DAES_ASM -DVP8_ASM -DWHIRLPOOL_ASM -DGHASH_ASM
The 'numbers' are in 1000s of bytes per second processed.
type          16 bytes      64 bytes      256 bytes     1024 bytes     8192 bytes
aes-128 cbc    83587.78k    94327.79k    97410.47k    98817.03k    98631.68k
aes-192 cbc    71229.27k    79341.12k    81712.47k    81968.13k    82561.11k
aes-256 cbc    62115.35k    68170.56k    69711.19k    70144.34k    70216.36k
[04/14/19]seed@VM:~/.../Task5$
```

Benchmarking on AES

The times between my trials of both RSA and AES are similar to the openssl speed tests

Task 6: Create Digital Signature:

For this task I created the example.txt file that has content below:



The steps demonstrate how I used SHA256 to hash and signed the file:

First, I generate the private key using RSA algorithm:

Khanh Nguyen
525000335
CSCE465

```
/bin/bash 80x24
[04/19/19]seed@VM:~/.../Task6$
Evaluation-SEED$ openssl genpkey -algorithm rsa -out private_key.pem
```

After that, I generate the public key associated with the private key:

```
/bin/bash 80x24
[04/19/19]seed@VM:~/.../Task6$
Evaluation-SEED$ openssl pkey -in private_key.pem -out public_key.pem -pubout
```

This is the content of public key:

```
public_key.pem
~/Desktop/HW5/Task6

File Edit View Search Tools Documents Help
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQCzPRnhQRfMZ3GiQ2F2Z1mJeJ+A
nXwkLKN97BgrZASAsz68PvheDiRR8MLQM7T7eCLF8j2qdwW6rYaSXoWEYjiHdYUj
X2WpwE0PobrrZeyRS0Bd7aKcIxxb/L+QtLYFRWDP2gkrEFkbYNNAwCbgp8ve3Qs
rg9rUbDpyho2RX2ZqwIDAQAB
-----END PUBLIC KEY-----
```

Next, I generated the hash value and signed the plain text example.txt using SHA256:

```
Evaluation-SEED$ openssl dgst -sha256 -sign private_key.pem -out enc_sha256_example.txt example.txt
```

To verify the file, I used the following command:

```
Evaluation-SEED$ openssl dgst -sha256 -verify public_key.pem -signature example.sha256 example.txt
Verified OK
[04/14/19]seed@VM:~/.../Task6$
```

It shows that the Verification is complete.

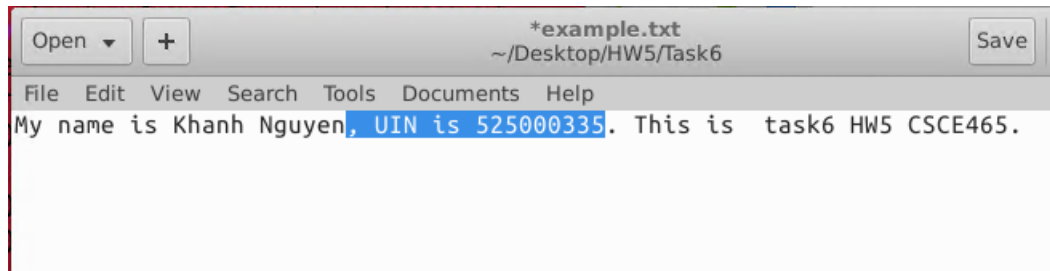
To observe that if the digital signature can still work if I modify the file, I change the content of the example.txt a little as shown below:

```
example.txt
~/Desktop/HW5/Task6

File Edit View Search Tools Documents Help
My name is Khanh Nguyen. This is task6 HW5 CSCE465.
```

Original content

Khanh Nguyen
525000335
CSCE465



Modified content

After that I ran the code to verify the signature again but I the verification failed.

```
Evaluation-SEED$ openssl dgst -sha256 -verify public_key.pem -signature example  
.sha256 example.txt  
Verification Failure  
[04/14/19]seed@VM:~/.../Task6$
```

It shows that if the content of the file changes, the public key private key can also change. It leads to the result of verification failure. Digital signatures are important in security because they ensure integrity and non-repudiation. Using this, we can tell if a document was tampered, and we know not to use it. Furthermore, digital signatures help ensure delivery and helps track who interacts with the information.