Assignment 4 – ECE 1155

Rayan Hassan 4511021

Question 1

The terminal sends a random R, the card gets it and it gets encrypted (signed) using the card's private key. Then, the terminal will decrypt it with the card's public key and it verifies the signature (if the received R is the same one then the card is authentic).

Question 2

1) The issuer is "Symantec Corporation"

2) The owner is "National Science Foundation"

3) Symantec generated the signature. This signature can be verified using the CA's public key.

4) The encryption method used for the key is RSA. e = 65537 (in decimal) and n is a large number beginning with 00:ca:fb:26 (first four bytes)

5) The private key is not included in the certificate

6) The signature algorithm in the certificate is sha256withRSAEncryption. The encryption method is RSA. The hash function used is SHA256

Question 3

1) This is called a chain of trust. The reason it's used in a nutshell is to allow other users to create and use software. More specifically, the chain starts with a root CA that vouches for intermediate CA and sign their certificate. Then, intermediate CA can issue certificate for other users (subordinate CA).

2) The root CA has a self-signed certificate.

Question 4

1) Cryptographic hash functions can be applied to block of data of any size. Also, it should produce output of fixed length. It should relatively be easy to compute. It has the one-way property (computationally infeasible to invert the hash, meaning that given H(m) it is hard to get m. The final property is collision resistant: weak collision resistant meaning that given m_1, it is computationally infeasible to find $m_2 \neq m_1$ such that $H(m_2) = H(m_1)$. And strong collision resistant, meaning it is computationally infeasible to find $m_2 \neq m_1$ such that $H(m_2) = H(m_1)$.

2) The first three properties are satisfied. In fact, the hash function can be applied to block of data of any size since we have a fixed value of n and the summation of the elements a_i. Also, the output will have a fixed size since n is fixed (so result will go from 0 to n-1). Moreover, the hash function is easy to calculate. However, it is not computationally infeasible to find m from H(m) since we have a fixed value n (so limited range of results). Similarly, it is easy to find to data sets that are not the same but result in the same result. For example let's take n=9, A: {4,3,1} and B:{11,9,6}. (4+3+1)mod9 = 8 mod 9 = 8.

(11+9+6) mod 9 = 26 mod9 = 8.

So one-way property and collision resistant property are not satisfied. So this is not a valid cryptographic hash.

Question 5

Hash collision might occur because hash functions can have inputs with infinite length and a defined output length. Therefore, there is a possibility that two different inputs produce the same output hash.

Question 6

1) The Birthday paradox is basically trying to find how many people we should have in a room to have a 50% probability that at least 2 have the same birthday.

2) Statistics have shown that we only need 23 people in a room in order for us to have a probability of 0.5 of having 2 with the same birthday. If we “map” this concept to our hash functions, it shows that it only takes 23 inputs before we have a 50% chance of having a collision, which is bad. So here it all depends on how large our hash is: the larger, the better.

3) For N bits input, we need to compute $2^{N/2}$ hashes before we get a 50% chance of getting a collision.

4) We need to compute $\sqrt{M2^N}$ hashes to find M collisions.

5) A basic example is if Bob wants Alice to sign an evil message E. Bob would create multiple innocent message L that Alice won’t mind to sign. Now Bob creates variants of L and E with same meaning as L and E respectively, but minor differences. Bob can find variants of evil message (Ei) and variants of innocent message (Li) with the same hash. Finally, Bob sends (Li) to Alice, she signs the hash. Then, Bob validates the evil message E using the hash.

6) For the hash to be more secure, it should be very long.

Question 7

1) MD stands for Message digest and SHA stands for Secure Hash Algorithm.

2) Basic functions used in SHA are AND, OR, NOT, XOR and circular shifts performed in every round. And in the end there is a word by word addition mod 2^{64}

3) Examples of hash algorithms are MD5 resulting in 128 bits hash value, and SHA256 with 256 bit hash value

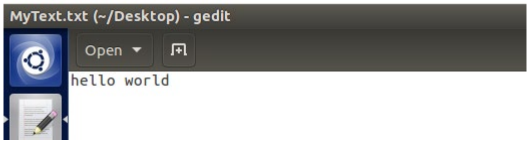
Question 8

1) HMAC is used if let’s say two entities want to communicate with each other privately and so they need a way to verify that the packets they received are authentic. So HMAC is used to verify authenticity and integrity of a message. It uses cryptographic hash functions coupled with a secret key.

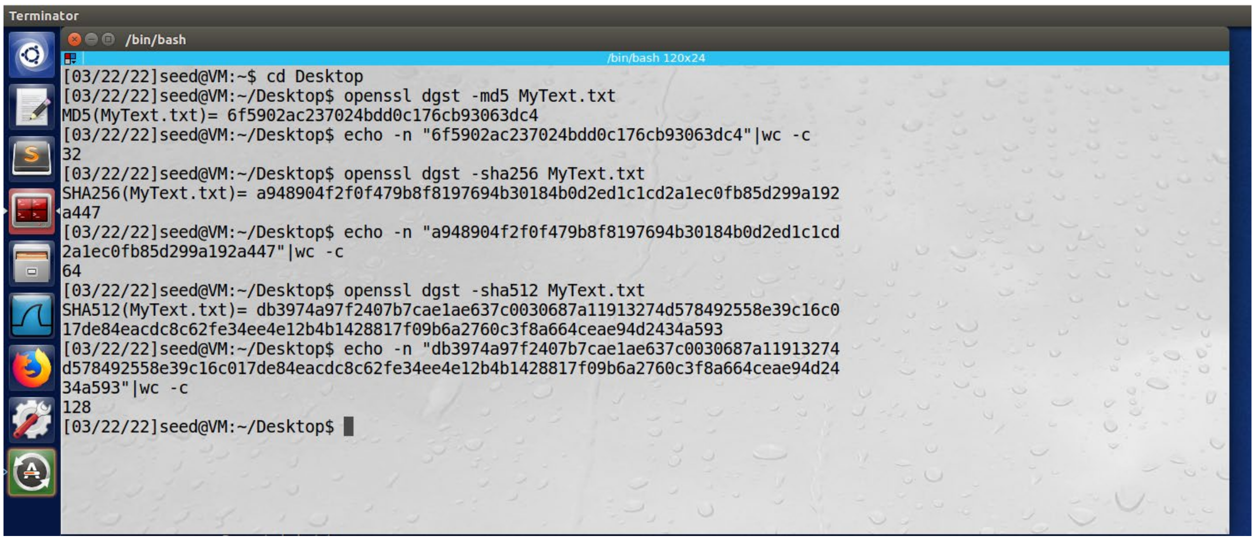
2) Basically the sender evaluates hash from the message. Instead of doing any encryption, if input to the hash algorithm is different, he simply appends the key to the message. Since the key is different for every device we will get a hash that is function of the message and the key. Similarly, the receiver will get the message and repeat the process but locally, to re-evaluate the hash. Then he compares it to hash appended to the message by the sender. If they’re the same then the integrity validation passes.

Lab part

Task1

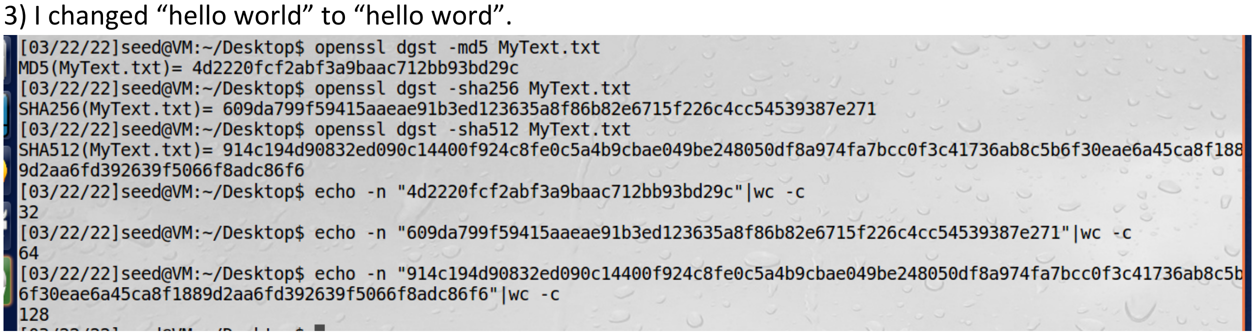
1) 

2)



I found the MD5, SHA-256, SHA-512 in the command window, each directly followed by their respective lengths. So the lengths of the output hash for each are 32, 64 and 128 bytes for MD5, SHA-256 and SHA-512 respectively.

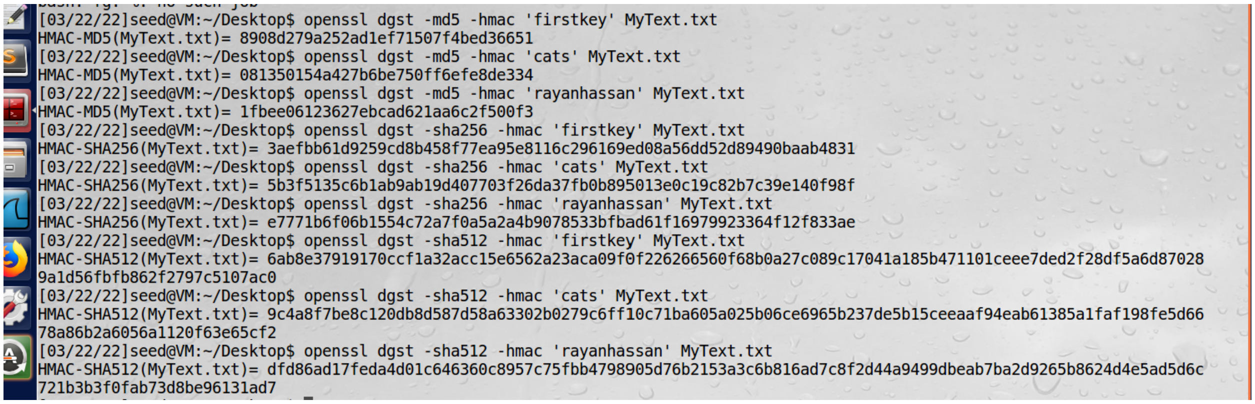
3) I changed “hello world” to “hello word”.



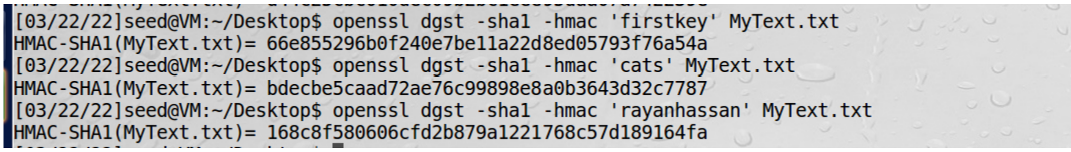
As we can see, the output hash changes drastically for each type.

Task 2

a) The text file is “Hello world”



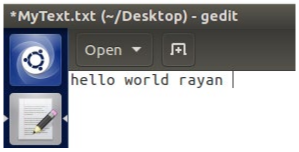
```
[03/22/22]seed@VM:~/Desktop$ openssl dgst -md5 -hmac 'firstkey' MyText.txt
HMAC-MD5(MyText.txt)= 8908d279a252ad1ef71507f4bed36651
[03/22/22]seed@VM:~/Desktop$ openssl dgst -md5 -hmac 'cats' MyText.txt
HMAC-MD5(MyText.txt)= 081350154a427b6be750ff6efe8de334
[03/22/22]seed@VM:~/Desktop$ openssl dgst -md5 -hmac 'rayanhassan' MyText.txt
HMAC-MD5(MyText.txt)= 1fbee06123627ebcad621aa6c2f500f3
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha256 -hmac 'firstkey' MyText.txt
HMAC-SHA256(MyText.txt)= 3aefbb61d9259cd8b458f77ea95e8116c296169ed08a56dd52d89490baab4831
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha256 -hmac 'cats' MyText.txt
HMAC-SHA256(MyText.txt)= 5b3f5135c6b1ab9ab19d407703f26da37fb0b895013e0c19c82b7c39e140f98f
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha256 -hmac 'rayanhassan' MyText.txt
HMAC-SHA256(MyText.txt)= e7771b6f06b1554c72a7f0a5a2a4b9078533bfbad61f16979923364f12f833ae
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha512 -hmac 'firstkey' MyText.txt
HMAC-SHA512(MyText.txt)= 6ab8e37919170ccf1a32acc15e6562a23aca09f0f226266560f68b0a27c089c17041a185b471101ceee7ded2f28df5a6d87028
9a1d56fbfb862f2797c5107ac0
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha512 -hmac 'cats' MyText.txt
HMAC-SHA512(MyText.txt)= 9c4a8f7be8c120db8d587d58a63302b0279c6ff10c71ba605a025b06ce6965b237de5b15ceeaaf94eab61385a1faf198fe5d66
78a86b2a6056a1120f63e65cf2
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha512 -hmac 'rayanhassan' MyText.txt
HMAC-SHA512(MyText.txt)= dfd86ad17feda4d01c646360c8957c75fbb4798905d76b2153a3c6b816ad7c8f2d44a9499dbeab7ba2d9265b8624d4e5ad5d6c
721b3b3f0fab73d8be96131ad7
```

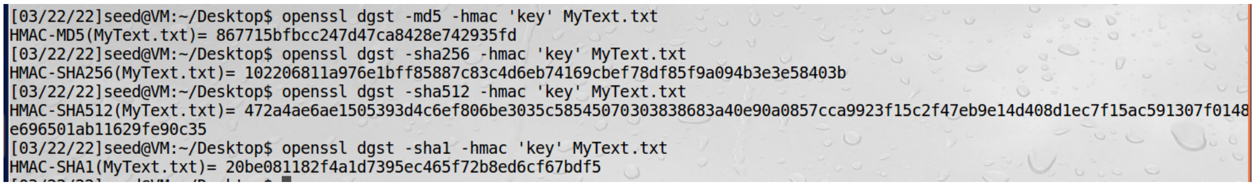


```
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha1 -hmac 'firstkey' MyText.txt
HMAC-SHA1(MyText.txt)= 66e855296b0f240e7be11a22d8ed05793f76a54a
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha1 -hmac 'cats' MyText.txt
HMAC-SHA1(MyText.txt)= bdecbe5caad72ae76c99898e8a0b3643d32c7787
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha1 -hmac 'rayanhassan' MyText.txt
HMAC-SHA1(MyText.txt)= 168c8f580606cfd2b879a1221768c57d189164fa
```

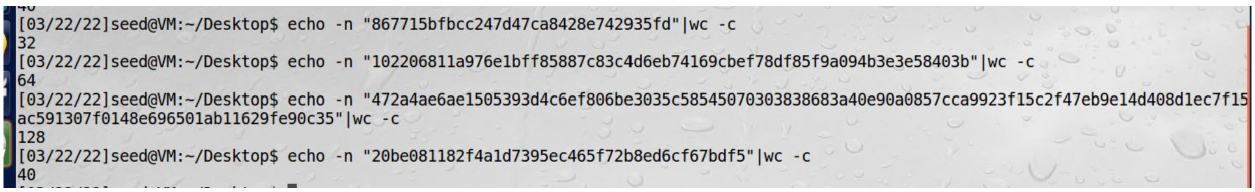
We don’t need a key with a fixed sized because it will add padding to the key and therefore at the end we’re going have the same length.

b) First text file (short)



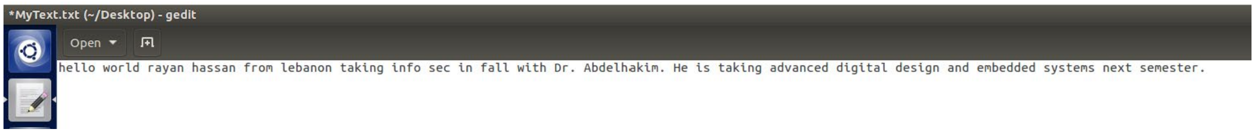


```
[03/22/22]seed@VM:~/Desktop$ openssl dgst -md5 -hmac 'key' MyText.txt
HMAC-MD5(MyText.txt)= 867715bfbcc247d47ca8428e742935fd
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha256 -hmac 'key' MyText.txt
HMAC-SHA256(MyText.txt)= 102206811a976e1bff85887c83c4d6eb74169cbef78df85f9a094b3e3e58403b
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha512 -hmac 'key' MyText.txt
HMAC-SHA512(MyText.txt)= 472a4ae6ae1505393d4c6ef806be3035c585450703038386 83a40e90a0857cca9923f15c2f47eb9e14d408d1ec7f15ac591307f0148
e696501ab11629fe90c35
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha1 -hmac 'key' MyText.txt
HMAC-SHA1(MyText.txt)= 20be081182f4a1d7395ec465f72b8ed6cf67bdf5
```



```
[03/22/22]seed@VM:~/Desktop$ echo -n "867715bfbcc247d47ca8428e742935fd"|wc -c
32
[03/22/22]seed@VM:~/Desktop$ echo -n "102206811a976e1bff85887c83c4d6eb74169cbef78df85f9a094b3e3e58403b"|wc -c
64
[03/22/22]seed@VM:~/Desktop$ echo -n "472a4ae6ae1505393d4c6ef806be3035c58545070303838683a40e90a0857cca9923f15c2f47eb9e14d408d1ec7f15
ac591307f0148e696501ab11629fe90c35"|wc -c
128
[03/22/22]seed@VM:~/Desktop$ echo -n "20be081182f4a1d7395ec465f72b8ed6cf67bdf5"|wc -c
40
```

Second text file (long)



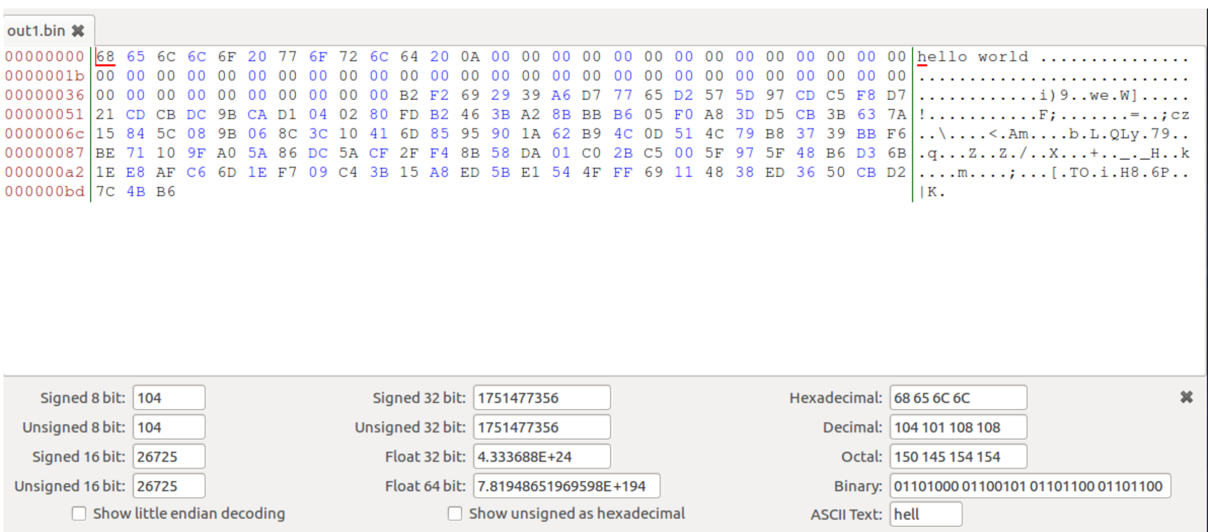
```
[03/22/22]seed@VM:~/Desktop$ openssl dgst -md5 -hmac 'key' MyText.txt
HMAC-MD5(MyText.txt)= 18eafbf552f53f695f6807d7178a6cec
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha256 -hmac 'key' MyText.txt
HMAC-SHA256(MyText.txt)= 70f1735af04366913b4950543671725080c5ca15a7fe4b0f298d7e7f1d26d8dc
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha512 -hmac 'key' MyText.txt
HMAC-SHA512(MyText.txt)= ea2b71cf597fa6f9fc0d83849f045a1ad2c32a880424225c4f4428b977e9239431c700216f36cfa20fc09c6e98df5a02519ede779a4
82407299222ba28c9aa28
[03/22/22]seed@VM:~/Desktop$ openssl dgst -sha1 -hmac 'key' MyText.txt
HMAC-SHA1(MyText.txt)= dd169e3804c56b27053f075352f9d4c143ec4118
```

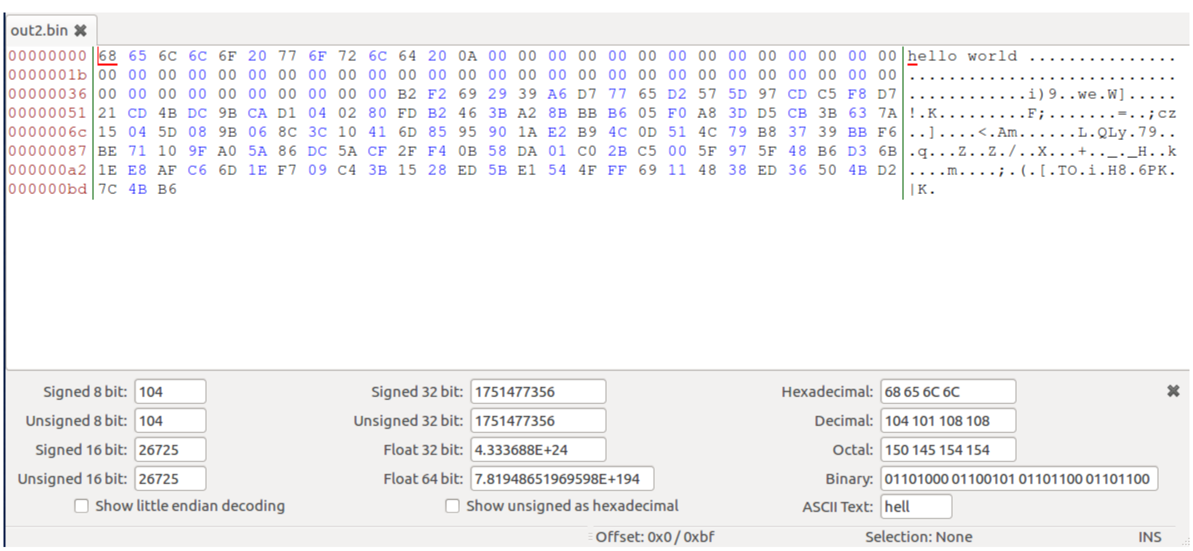
```
[03/22/22]seed@VM:~/Desktop$ echo -n "18eafbf552f53f695f6807d7178a6cec"|wc -c
32
[03/22/22]seed@VM:~/Desktop$ echo -n "70f1735af04366913b4950543671725080c5ca15a7fe4b0f298d7e7f1d26d8dc"|wc -c
64
[03/22/22]seed@VM:~/Desktop$ echo -n "ea2b71cf597fa6f9fc0d83849f045a1ad2c32a880424225c4f4428b977e9239431c700216f36cfa20fc09c6e98df5a
02519ede779a482407299222ba28c9aa28"|wc -c
128
[03/22/22]seed@VM:~/Desktop$ echo -n "dd169e3804c56b27053f075352f9d4c143ec4118"|wc -c
40
[03/22/22]seed@VM:~/Desktop$
```

The lengths of the keyed hash are the same whether the text file is long or short (because of the padding).

Task 3

The file I used is “Hello World”. Here are the output files I got.



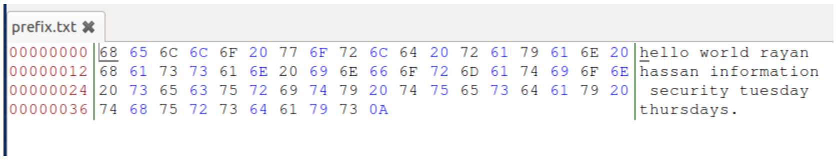


Question 1

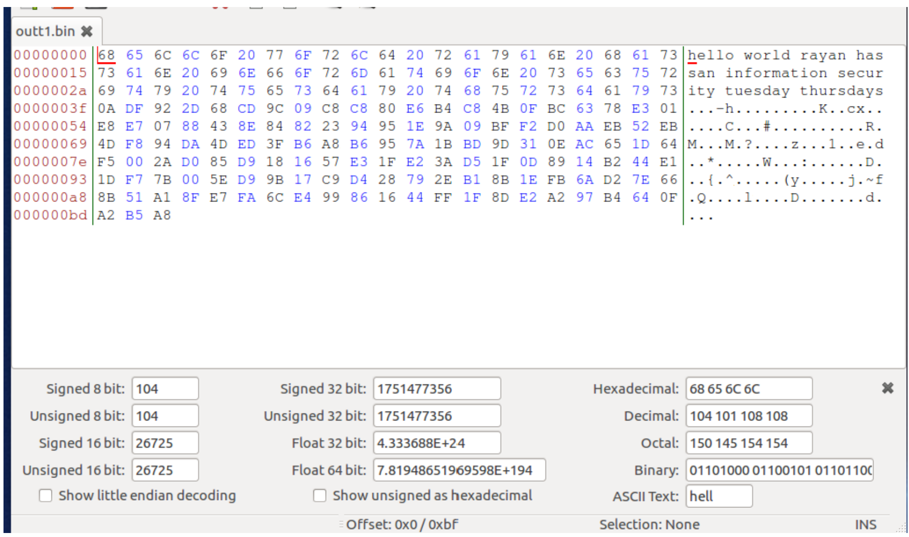
If the length of my prefix is not 64, like in the above case, it will add padding to get same length.

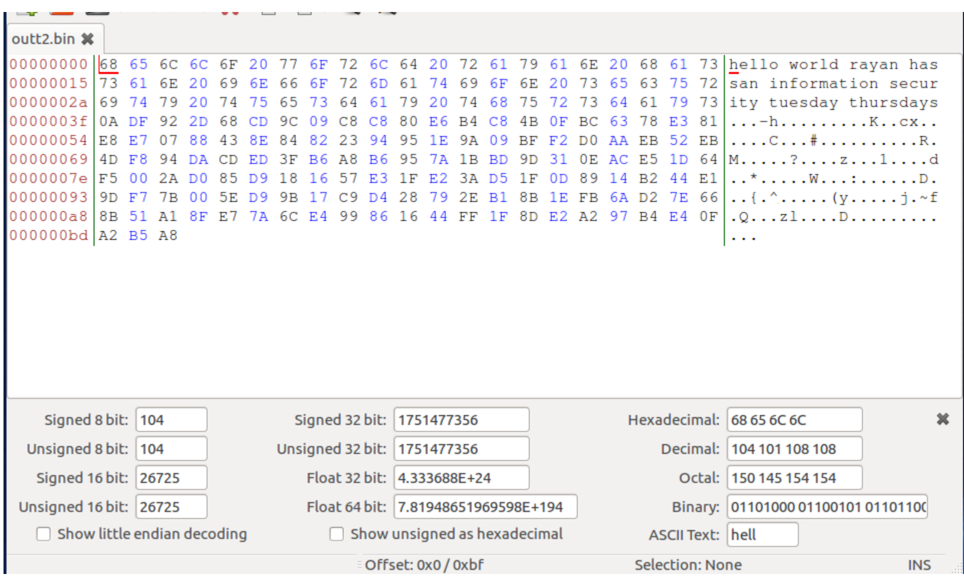
Question 2

This is my 64 bytes' prefix file



The resulting output files are the following





Here we see that the output files are slightly different. The bytes that are different are the 110th byte (changes from 'M' in outt1.bin to '.' in outt2.bin), the 124th byte (from 'e' in outt1.bin to '.' In outt2.bin), the 173th byte (from '.' In outt1.bin to 'z' in outt2.bin) and the 187th byte (from 'd' in outt1.bin to '.' In outt2.bin).

Task 4

GlobalSign Root CA
Identity: GlobalSign Root CA
Verified by: GlobalSign Root CA
Expires: 01/28/2028

Details

Subject Name
C (Country): BE
O (Organization): GlobalSign nv-sa
OU (Organizational Unit): Root CA
CN (Common Name): GlobalSign Root CA

Issuer Name
C (Country): BE
O (Organization): GlobalSign nv-sa
OU (Organizational Unit): Root CA
CN (Common Name): GlobalSign Root CA

Issued Certificate
Version: 3
Serial Number: 04 00 00 00 00 01 15 4B 5A C3 94
Not Valid Before: 1998-09-01
Not Valid After: 2028-01-28

Certificate Fingerprints
SHA1: B1 BC 96 8B D4 F4 9D 62 2A A8 9A 81 F2 15 01 52 A4 1D 82 9C
MD5: 3E 45 52 15 09 51 92 E1 B7 5D 37 9F B1 87 29 8A

Public Key Info
Key Algorithm: RSA
Key Parameters: 05 00
Key Size: 2048
Key SHA1 Fingerprint: 87 DB D4 5F B0 92 8D 4E 1D F8 15 67 E7 F2 AB AF D6 2B 67 75
Public Key: 30 82 01 0A 02 82 01 01 00 DA 0E E6 99 8D CE A3 E3 4F 8A 7E FB F1 8B 83 25 6B EA 48 1F F1 2A B0 B9 95 11 04 BD F0 63 D1 E2 67 66 CF 1C DD CF 1B 48 2B EE 8D 89 8E 9A AF 29 80 65 AB E9 C7 2D 12 CB AB 1C 4C 70 07 A1 3D 0A 30 CD 15 8D 4F F8 DD D4 8C 50 15
1C EF 50 EE C4 2E F7 FC E9 52 F2 91 7D E0 6D D5 35 30 8E 5E 43 73 F2 41 E9 D5 6A E3 B2 89 3A 56 39 38 6F 06 3C 88 69 5B 2A 4D C5 A7 54 B8 6C 89 CC 9B F9 3C CA E5 FD 89 F5 12 3C 92 78 96 D6 DC 74 6E 93 44 61 D1 8D C7 46 B2 75 0E 86 E8 19 8A D5 6D 6C D5
78 16 95 A2 E9 C8 0A 38 EB F2 24 13 4F 73 54 93 13 85 3A 1B BC 1E 34 B5 8B 05 8C B9 77 8B B1 DB 1F 20 91 AB 09 53 6E 90 CE 7B 37 74 B9 70 47 91 22 51 63 16 79 AE B1 AE 41 26 08 C8 19 2B D1 46 AA 48 D6 64 2A D7 83 34 FF 2C 2A C1 6C 19 43 4A 07 85 E7 D3
7C F6 21 68 EF EA F2 52 9F 7F 93 90 CF 02 03 01 00 01

Key Usage
Usages: Digital signature
Critical: Yes

Basic Constraints
Certificate Authority: Yes
Max Path Length: Unlimited
Critical: Yes

Subject Key Identifier
Key Identifier: 60 7B 66 1A 45 0D 97 CA 89 50 2F 7D 04 CD 34 A8 FF FC FD 4B
Critical: No

Signature
Signature Algorithm: SHA1 with RSA
Signature Parameters: 05 00
Signature: D6 73 E7 7C 4F 76 D0 8D BF EC BA A2 BE 34 C5 28 32 B5 7C FC 6C 9C 2C 2B BD 09 9E 53 BF 6B 5E AA 11 48 B6 E5 08 A3 B3 CA 3D 61 4D D3 46 09 B3 3E C3 A0 E3 63 55 1B F2 BA EF AD 39 E1 43 B9 38 A3 E6 2F 8A 26 3B EF A0 50 56 F9 C6 0A FD 38 CD C4 0B 70 51 94
97 98 04 DF C3 5F 94 D5 15 C9 14 41 9C C4 5D 75 64 15 0D FF 55 30 EC 86 8F FF 0D EF 2C B9 63 46 F6 AA FC DF BC 69 FD 2E 12 48 64 9A E0 95 F0 A6 EF 29 8F 01 B1 15 B5 0C 1D A5 FE 69 2C 69 24 78 1E B3 A7 1C 71 62 EE CA C8 97 AC 17 5D 8A C2 F8 47 86 6E 2A
C4 56 31 95 D0 67 89 85 2B F9 6C A6 5D 46 9D 0C AA 82 E4 99 51 DD 70 B7 DB 56 3D 61 E4 6A E1 5C D6 F6 FE 3D DE 41 CC 07 AE 63 52 BF 53 53 F4 2B E9 C7 FD B6 F7 82 5F 85 D2 41 18 DB 81 B3 04 1C C5 1F A4 80 6F 15 20 C9 DE 0C 88 0A 1D D6 66 55 E2 FC 48 C9
29 26 69 E0

a) The hash algorithm used is SHA1 and RSA for encryption.

b) The public key is 2048 bit

c) The public key algorithm used is RSA

d) Verification:

```
[03/22/22]seed@VM:~/Desktop$ openssl verify GlobalSignRootCA.crt
GlobalSignRootCA.crt: OK
[03/22/22]seed@VM:~/Desktop$ 
```