Patrick Korianski

Project 1, 4095: Introduction to Network Security

Late Days Used: 1 day

Assignment:

1. **What version of OpenSSL are you using?**

   OpenSSL 1.0.1f 6 Jan 2014

2. **Consider the following 64 bit key: 0xe0e0e0e0f1f1f1f1. Encrypt the following sentence**
   The quick brown fox jumped over the lazy dog
   Note that there is no "." at the end of the sentence. This is intentional.

2a. Use DES in electronic codebook mode (15 points)

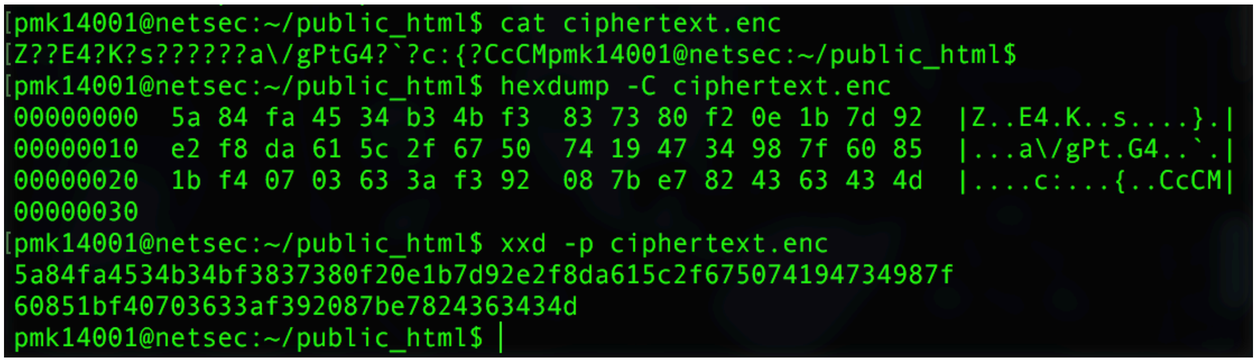
2i. What command did you use to generate ciphertext?

To generate the ciphertext in DES-ECB I used the following command (plaintext.txt is the text file I created to store the sentence in):

```
openssl enc -des-ecb -K e0e0e0e0f1f1f1f1 -in plaintext.txt -out ciphertext.enc
```

2ii. What is a hexadecimal representation of the resulting ciphertext?

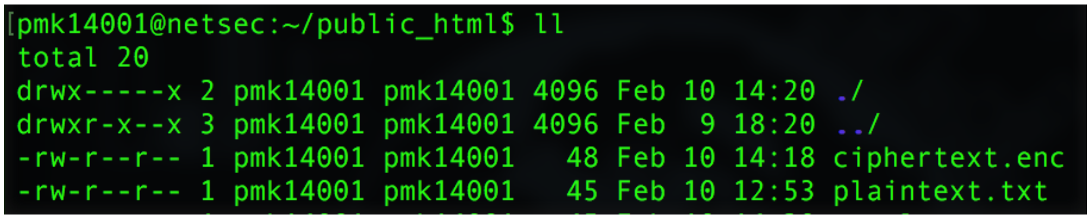
Below, shows the two ways I used to convert to the hexadecimal representation the first way is using hexdump and the second way is using xxd. I used cat ciphertext.enc to show the output of the original output of the ciphertext file.



```
[pmk14001@netsec:~/public_html$ cat ciphertext.enc
[Z??E4?K?s??????a\/gPtG4?`?c:{?CcCMpmk14001@netsec:~/public_html$
[pmk14001@netsec:~/public_html$ hexdump -C ciphertext.enc
00000000  5a 84 fa 45 34 b3 4b f3  83 73 80 f2 0e 1b 7d 92  |Z..E4.K..s....}.|
00000010  e2 f8 da 61 5c 2f 67 50  74 19 47 34 98 7f 60 85  |...a\/gPt.G4..`.|
00000020  1b f4 07 03 63 3a f3 92  08 7b e7 82 43 63 43 4d  |....c:...{..CcCM|
00000030
[pmk14001@netsec:~/public_html$ xxd -p ciphertext.enc
5a84fa4534b34bf3837380f20e1b7d92e2f8da615c2f675074194734987f
60851bf40703633af392087be7824363434d
pmk14001@netsec:~/public_html$ |
```

2iii. What is output length of the ciphertext?

The length is shown as 48MB as shown below



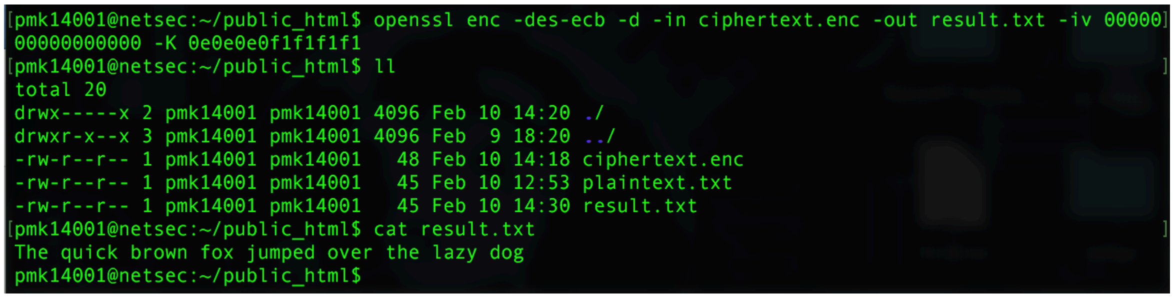
```
[pmk14001@netsec:~/public_html$ ll
 total 20
 drwx-----x 2 pmk14001 pmk14001 4096 Feb 10 14:20 ./
 drwxr-x--x 3 pmk14001 pmk14001 4096 Feb  9 18:20 ../
 -rw-r--r-- 1 pmk14001 pmk14001   48 Feb 10 14:18 ciphertext.enc
 -rw-r--r-- 1 pmk14001 pmk14001   45 Feb 10 12:53 plaintext.txt
```

2iv. Why?

The length of the ciphertext should be relatively close to the size of the plaintext which it is only 3 off. If you used the –a command in the encryption, the ciphertext.enc would be around 48 bits in which I had tried an example and got 48.

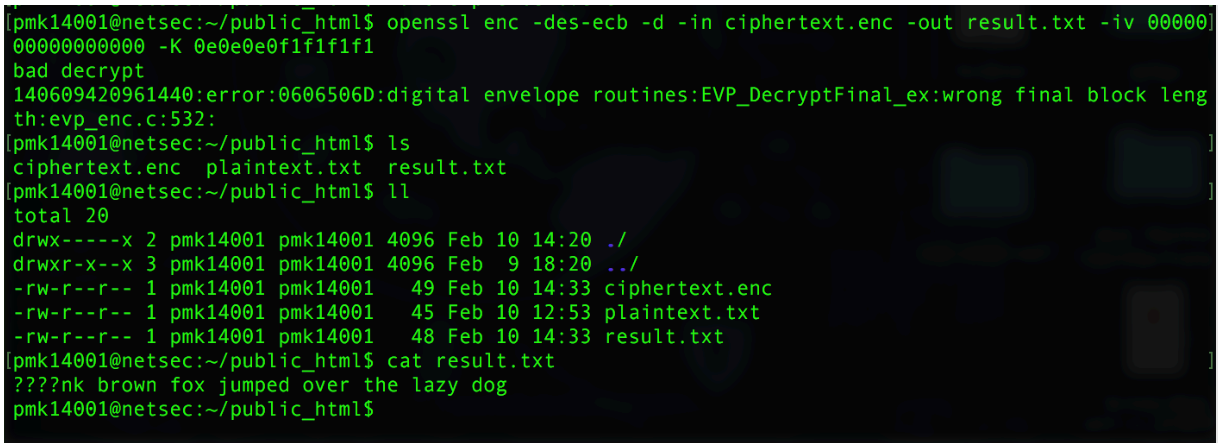
2v. What command should you use to decrypt the ciphertext?

To decrypt the ciphertext, you will need to add the –d command to the command line and also use the in as –in ciphertext.enc and out as –out <textfile name>. As you will see, both the result.txt and the plaintext.txt are the same size and I used the cat command to make sure it was the same sentence. My result for decrypting is shown below.



```
[pmk14001@netsec:~/public_html$ openssl enc -des-ecb -d -in ciphertext.enc -out result.txt -iv 00000]
 00000000000 -K 0e0e0e0f1f1f1f1
[pmk14001@netsec:~/public_html$ ll                                                              ]
 total 20
 drwx-----x 2 pmk14001 pmk14001 4096 Feb 10 14:20 ./
 drwxr-x--x 3 pmk14001 pmk14001 4096 Feb  9 18:20 ../
 -rw-r--r-- 1 pmk14001 pmk14001   48 Feb 10 14:18 ciphertext.enc
 -rw-r--r-- 1 pmk14001 pmk14001   45 Feb 10 12:53 plaintext.txt
 -rw-r--r-- 1 pmk14001 pmk14001   45 Feb 10 14:30 result.txt
[pmk14001@netsec:~/public_html$ cat result.txt                                                  ]
 The quick brown fox jumped over the lazy dog
 pmk14001@netsec:~/public_html$
```

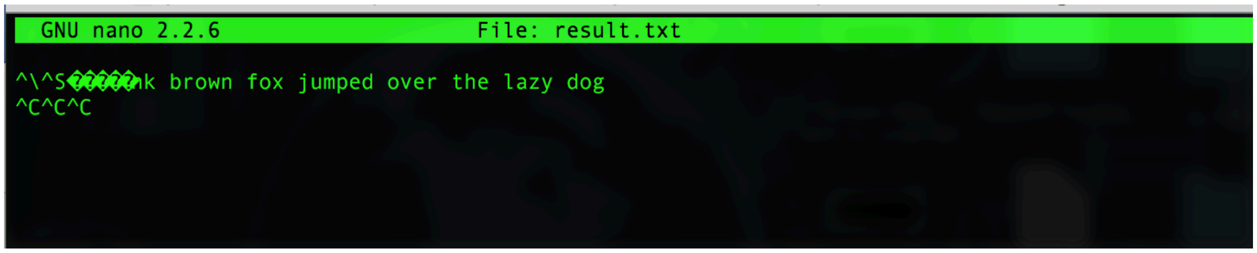
2vi. Change the first byte of the ciphertext file to a capital B. Decrypt this modified ciphertext. What is the resulting decryption?



```
[pmk14001@netsec:~/public_html$ openssl enc -des-ecb -d -in ciphertext.enc -out result.txt -iv 00000]
 00000000000 -K 0e0e0e0f1f1f1f1
 bad decrypt
 140609420961440:error:0606506D:digital envelope routines:EVP_DecryptFinal_ex:wrong final block leng
 th:evp_enc.c:532:
[pmk14001@netsec:~/public_html$ ls                                                               ]
 ciphertext.enc  plaintext.txt  result.txt
[pmk14001@netsec:~/public_html$ ll                                                               ]
 total 20
 drwx-----x 2 pmk14001 pmk14001 4096 Feb 10 14:20 ./
 drwxr-x--x 3 pmk14001 pmk14001 4096 Feb  9 18:20 ../
 -rw-r--r-- 1 pmk14001 pmk14001   49 Feb 10 14:33 ciphertext.enc
 -rw-r--r-- 1 pmk14001 pmk14001   45 Feb 10 12:53 plaintext.txt
 -rw-r--r-- 1 pmk14001 pmk14001   48 Feb 10 14:33 result.txt
[pmk14001@netsec:~/public_html$ cat result.txt                                                   ]
 ????nk brown fox jumped over the lazy dog
 pmk14001@netsec:~/public_html$
```

2vii. How many characters in the plaintext where changed by your modification? Why?

When looking at my results, the first two words were messed up. Instead of saying "the quick", it replaced those with "^\^\S????nk" and it also added "^C^C^C" to the end of my file. This all happened because changing anything in the ciphertext will result in the decryption to mess up some characters in bits since the original encryption's ciphertext was not the same as the modified one. This is definitely something that happens to a lot of hackers who try to obtain an incorrect ciphertext and try to decrypt the information.



```
  GNU nano 2.2.6                    File: result.txt

^\^S������nk brown fox jumped over the lazy dog
^C^C^C
```

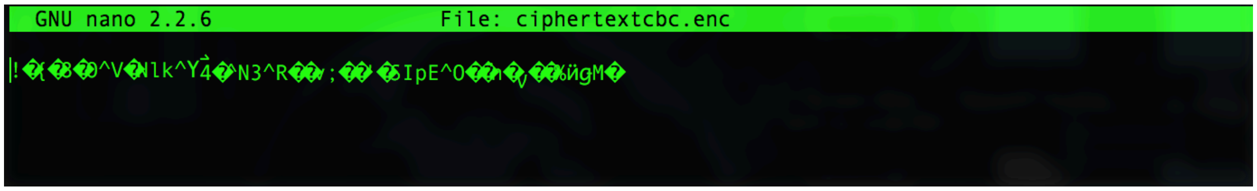
2b. Use DES in cipherblock chaining mode. Use the key from the previous part, use the following initialization vector 0x0123456789abcdef.

2i. What command did you use to generate ciphertext?

The command I used to generate the ciphertext was:

openssl enc -des-cbc -in plaintext.txt -out ciphertextcbc.enc -iv 0123456789abcdef -K e0e0e0e0f1f1f1f

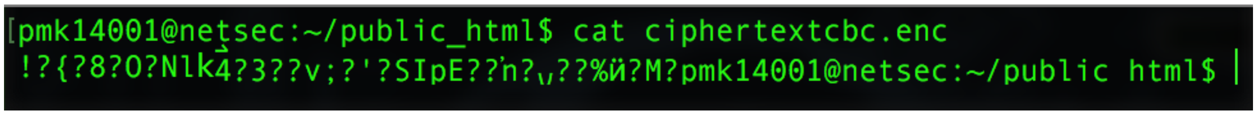
2ii. What is the resulting ciphertext?



```
  GNU nano 2.2.6                    File: ciphertextcbc.enc

!�{�8�O^V�Nlk^Y4�^N3^R��v;��'�SIpE^O��n�,��%йgM�
```

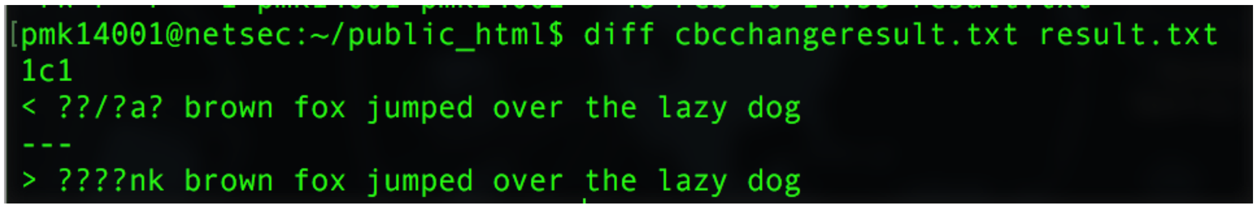
or



```
[pmk14001@netsec:~/public_html$ cat ciphertextcbc.enc
 !?{?8?O?Nlk4?3??v;?'?SIpE??n?,??%й?M?pmk14001@netsec:~/public html$ |
```

2iii. Change the first byte of ciphertext to a B. Decrypt the resulting ciphertext file. Compare the change to the previous part. What is different about the plaintext?

When changing the ciphertext file and comparing the results of both this decryption and the other decryption that used des-ecb, I got the following differences (cbcchangeresult.txt is using des-cbc and result was using des-ecb):



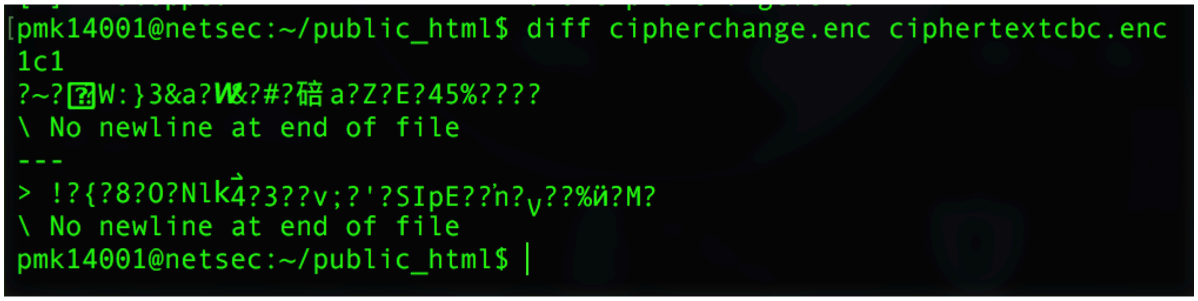
```
[pmk14001@netsec:~/public_html$ diff cbcchangeresult.txt result.txt
1c1
< ??/?a? brown fox jumped over the lazy dog
---
> ????nk brown fox jumped over the lazy dog
```

2iv. Look at the description of cipher block chaining, explain the behavior observed above.

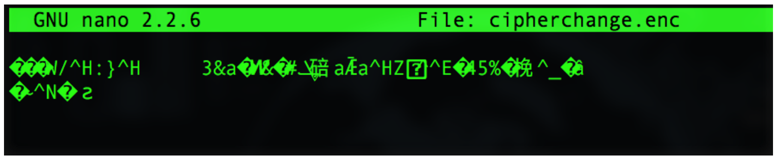
During Cipher-block chaining, the initialization vector(IV) is used in the first block where you first input your plaintext file. From there, it will go through the block cipher encryption which will then produce a ciphertext and that ciphertext will be used in the first block of the second round of encrypting the plaintext. In the electric code book mode, the IV is not used in any of the steps and the procedure goes from plaintext to the block cipher encryption to the ciphertext. To decrypt both is it basically the reverse order, so for CBC it starts with the cipher text, goes through the block cipher decryption then the IV is used on the following back with the plaintext. In EBC, it is directly the opposite, where it goes from ciphertext, to the block cipher decryption, and finally to the plaintext.

2v. Instead of changing a single character of the ciphertext, change the first character of the plaintext to B. What changes in the resulting ciphertext?

With just a single character change a lot had changed between the two ciphertext files which I will show below (cipherchange.enc is ciphertext for the changed plaintext and ciphertextcbc.enc is the ciphertext for the original plaintext). I will also show both in the nano text editor

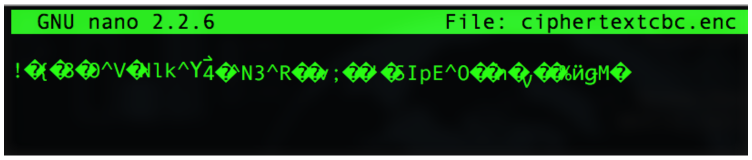


```
[pmk14001@netsec:~/public_html$ diff cipherchange.enc ciphertextcbc.enc
1c1
?~??W:}3&a?W&?#?碚a?Z?E?45%????
\ No newline at end of file
---
> !?{?8?O?Nlk4?3??v;?'?SIpE??n?v??%й?M?
\ No newline at end of file
pmk14001@netsec:~/public_html$ |
```



```
  GNU nano 2.2.6                      File: cipherchange.enc

���N/^H:}^H      3&a�W&�#_碚 aÆa^HZ?^E�45%�挽^_�a
�~^N� ƨ
```

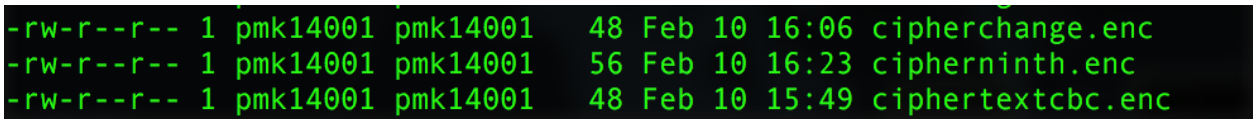


```
  GNU nano 2.2.6                      File: ciphertextcbc.enc

!�{�8�O^V�Nlk^Y4�^N3^R��;�� �SIpE^O��n�v��%йgM�
```

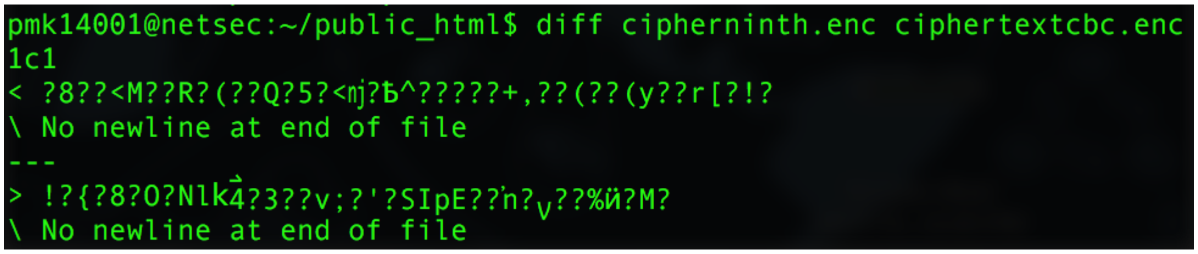
2vi. Revert your plaintext file back to the original sentence, instead of changing the first character, change the ninth character, the k in quick, to B. What has changed in the resulting ciphertext?

The first aspect I saw that really change was the length of the file. It went from 48 which the other two were to 56 which is a big jump in size as shown below (cipherninth is the results of the file that changed):

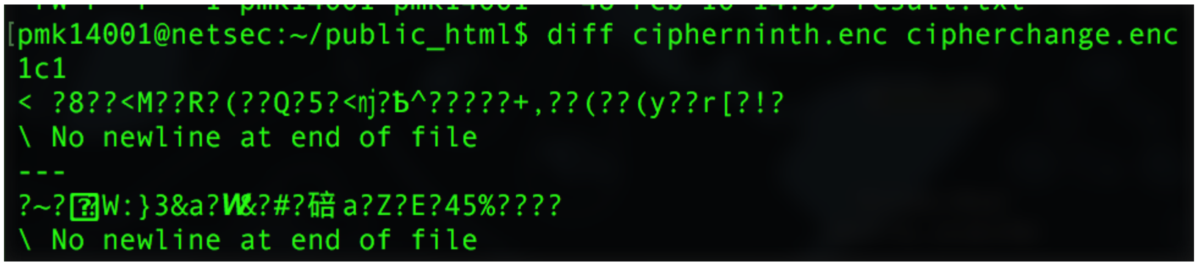


```
-rw-r--r-- 1 pmk14001 pmk14001   48 Feb 10 16:06 cipherchange.enc
-rw-r--r-- 1 pmk14001 pmk14001   56 Feb 10 16:23 cipherninth.enc
-rw-r--r-- 1 pmk14001 pmk14001   48 Feb 10 15:49 ciphertextcbc.enc
```

Second, running the diff command showed me the following differences between the other two ciphertext's (first photo is comparing to the original, second photo is comparing to the other changed plaintext):



```
pmk14001@netsec:~/public_html$ diff cipherninth.enc ciphertextcbc.enc
1c1
< ?8??<M??R?(??Q?5?<ǌ?Ѣ^?????+,??(??(y??r[?!?
\ No newline at end of file
---
> !?{?8?O?Nlk4?3??v;?'?SIpE??n?v??%й?M?
\ No newline at end of file
```



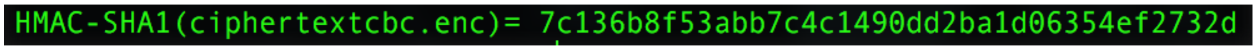
```
pmk14001@netsec:~/public_html$ diff cipherninth.enc cipherchange.enc
1c1
< ?8??<M??R?(??Q?5?<ǌ?Ѣ^?????+,??(??(y??r[?!?
\ No newline at end of file
---
?~??W:}3&a?W&?#?碚 a?Z?E?45%????
\ No newline at end of file
```

2c. We'll now add an integrity check to our ciphertext. We'll use SHA1 with the HMAC algorithm. Use the following MAC key: 0xfedcba9876543210. Your starting point will be good ciphertext in the previous part. Make sure you revert the change you made to the first character.

2i. What command did you use to generate a tag?

openssl dgst -sha1 -hmac fedcba9876543210 ciphertextcbc.enc

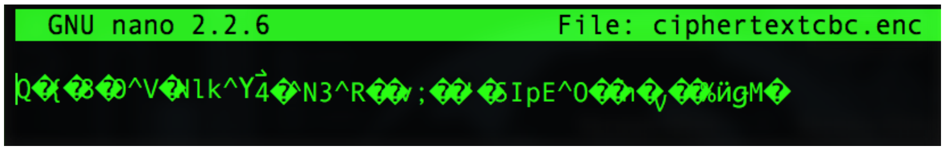
2ii. What is the tag that is created by the above command?



```
HMAC-SHA1(ciphertextcbc.enc)= 7c136b8f53abb7c4c1490dd2ba1d06354ef2732d
```

2iii. Change the first character of your ciphertext file. Recompute the tag value. What is the resulting tag?

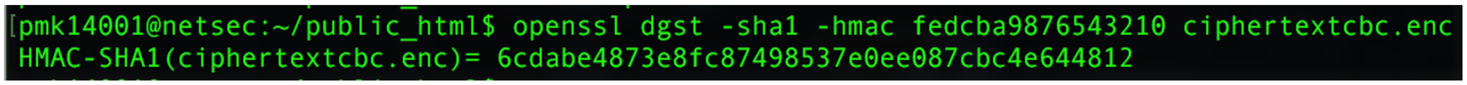
In my ciphertext file, I am replacing the first character with a Q as shown below:



```
  GNU nano 2.2.6                    File: ciphertextcbc.enc

Q�{�3�0^V�Nlk^Y4�^N3^R��v;�� �5IpE^O��n�v��%ӥgM�
```

The resulting tag for this modified ciphertext is shown below



```
[pmk14001@netsec:~/public_html$ openssl dgst -sha1 -hmac fedcba9876543210 ciphertextcbc.enc
HMAC-SHA1(ciphertextcbc.enc)= 6cdabe4873e8fc87498537e0ee087cbc4e644812
```

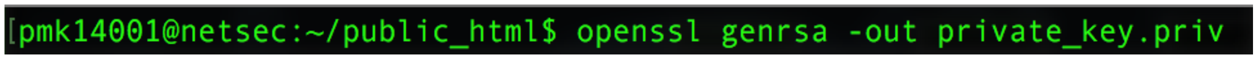
2iv. Given the above information, how could you combine an encryption scheme and a MAC algorithm that is resistant to active modification?

When looking through the different approaches of authenticated encryption, on the Wikipedia page, https://en.wikipedia.org/wiki/Authenticated_encryption, I believe using the MAC-then-Encryption (MtE) because it seemed to be least resistant to active modification. In, encrypt-and-MAC (E&M), it is said that some minor modifications to SSH is possible, and also in Encrypt-then-MAC(EtM).

3. We will now switch to using RSA cryptosystem.

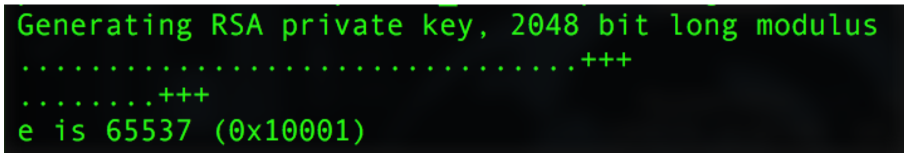
3a. Create a 2048 bit RSA private key using the genrsa command in openssl. Save this private key to private_key.priv

3i. What command did you use?



```
[pmk14001@netsec:~/public_html$ openssl genrsa -out private_key.priv
```

3ii. What is the public exponent e that is used?



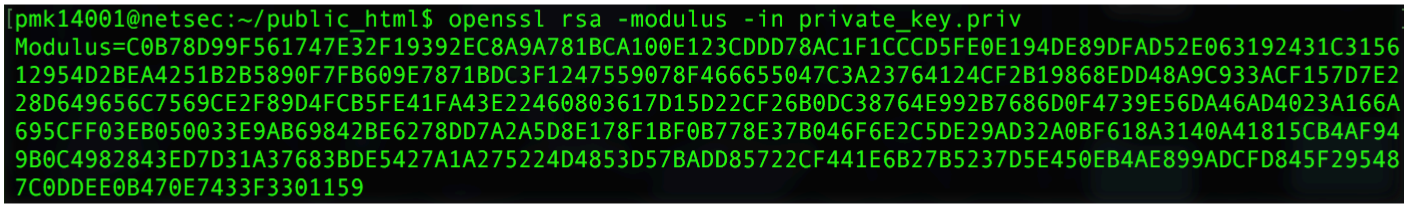
```
Generating RSA private key, 2048 bit long modulus
...............................+++
........+++
e is 65537 (0x10001)
```

3iii. Why is this exponent commonly used?

This key is commonly used because it is not a small number which are usually more vulnerable to a simple attack and can avoid these exponential attacks. Also, it used (2^16 +1) to obtain the number. Another reason it is commonly used is because it allows for efficient encryptions.

Sources: https://en.wikipedia.org/wiki/RSA_(cryptosystem), and Lecture 3-public-key-crypto

3iv. Use the openssl RSA command to write out the modulus n in hexadecimal format. What command did you use?

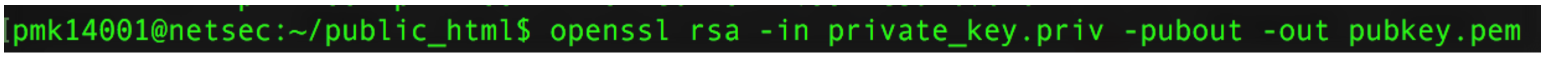


```
[pmk14001@netsec:~/public_html$ openssl rsa -modulus -in private_key.priv
Modulus=C0B78D99F561747E32F19392EC8A9A781BCA100E123CDDD78AC1F1CCCD5FE0E194DE89DFAD52E063192431C3156
12954D2BEA4251B2B5890F7FB609E7871BDC3F1247559078F466655047C3A23764124CF2B19868EDD48A9C933ACF157D7E2
28D649656C7569CE2F89D4FCB5FE41FA43E22460803617D15D22CF26B0DC38764E992B7686D0F4739E56DA46AD4023A166A
695CFF03EB050033E9AB69842BE6278DD7A2A5D8E178F1BF0B778E37B046F6E2C5DE29AD32A0BF618A3140A41815CB4AF94
9B0C4982843ED7D31A37683BDE5427A1A275224D4853D57BADD85722CF441E6B27B5237D5E450EB4AE899ADCFD845F29548
7C0DDEE0B470E7433F3301159
```

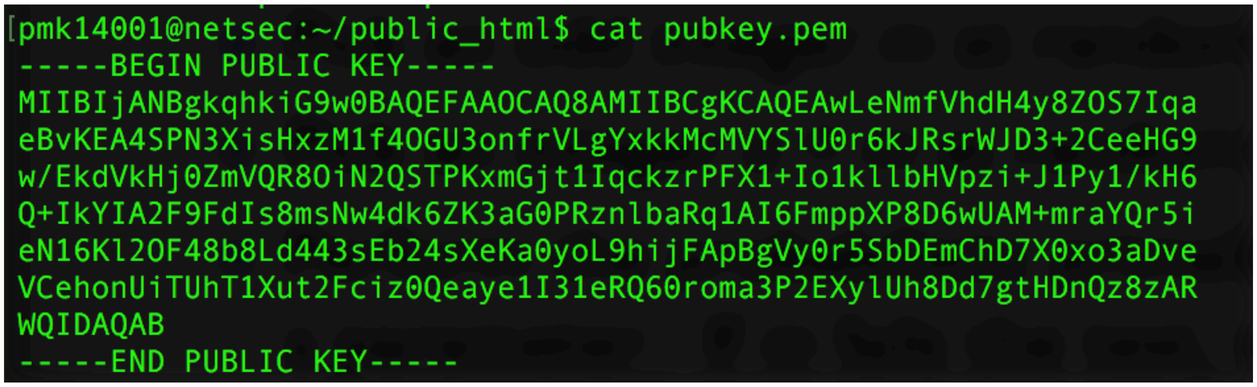
3v. How many bytes is the modulus encoded this way?

The modulus has 512 hexadecimal characters which is equal to 256 bytes (512/2).

3vi. Export the public key using the rsa command. What command did you use?



```
[pmk14001@netsec:~/public_html$ openssl rsa -in private_key.priv -pubout -out pubkey.pem
```

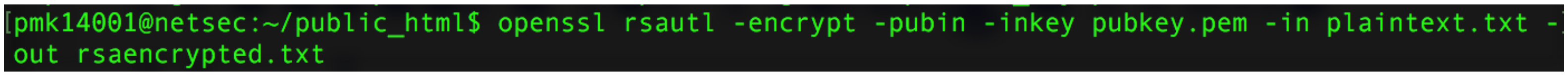


```
[pmk14001@netsec:~/public_html$ cat pubkey.pem
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAwLeNmfVhdH4y8ZOS7Iqa
eBvKEA4SPN3XisHxzM1f4OGU3onfrVLgYxkkMcMVYSlU0r6kJRsrWJD3+2CeeHG9
w/EkdVkHj0ZmVQR8OiN2QSTPKxmGjt1IqckzrPFX1+Io1kllbHVpzi+J1Py1/kH6
Q+IkYIA2F9FdIs8msNw4dk6ZK3aG0PRznlbaRq1AI6FmppXP8D6wUAM+mraYQr5i
eN16Kl2OF48b8Ld443sEb24sXeKa0yoL9hijFApBgVy0r5SbDEmChD7X0xo3aDve
VCehonUiTUhT1Xut2Fciz0Qeaye1I31eRQ60roma3P2EXylUh8Dd7gtHDnQz8zAR
WQIDAQAB
-----END PUBLIC KEY-----
```

3b. We'll now do some trial encryptions with RSA. For this part we'll use the rsautl command in openssl.

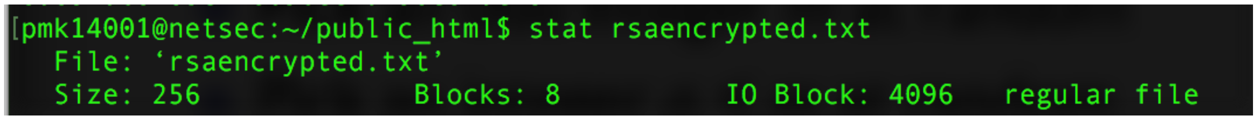
3i. Encrypt our standard plaintext using your RSA public key. What command do you use?

Below are the commands used to encrypt using a RSA public key. My public key was called "pubkey.pem" as shown below



```
[pmk14001@netsec:~/public_html$ openssl rsautl -encrypt -pubin -inkey pubkey.pem -in plaintext.txt -
out rsaencrypted.txt
```

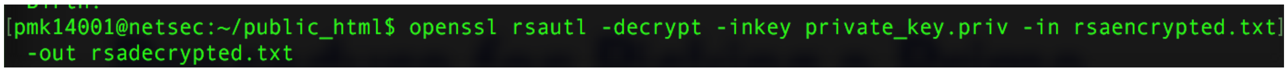
3ii. How long is the resulting ciphertext? Why?



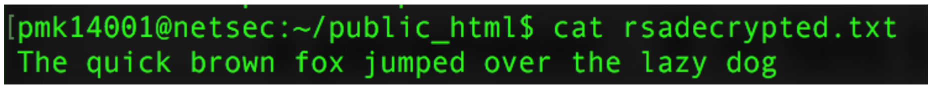
```
[pmk14001@netsec:~/public_html$ stat rsaencrypted.txt
  File: 'rsaencrypted.txt'
  Size: 256             Blocks: 8          IO Block: 4096   regular file
```

The resulting size of the ciphertext makes sense since the mod of the key generated in part 3a was equal to 256 bytes.

3iii. What is the corresponding command to decrypt the ciphertext? Verify that the returned value matches your starting plaintext.



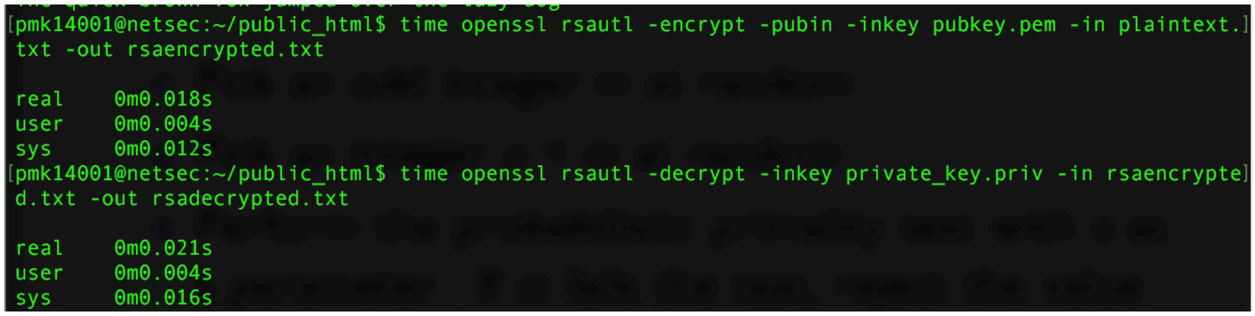
```
[pmk14001@netsec:~/public_html$ openssl rsautl -decrypt -inkey private_key.priv -in rsaencrypted.txt]
 -out rsadecrypted.txt
```



```
[pmk14001@netsec:~/public_html$ cat rsadecrypted.txt
 The quick brown fox jumped over the lazy dog
```

3iv. Time both the encrypt and decrypt operations using the UNIX time command. Report the user time, not the system or real time.

Using the time command in UNIX, the encryption took 0m0.004s in user time. Also, the decryption took 0m0.004s in user time. Both, can be shown below:



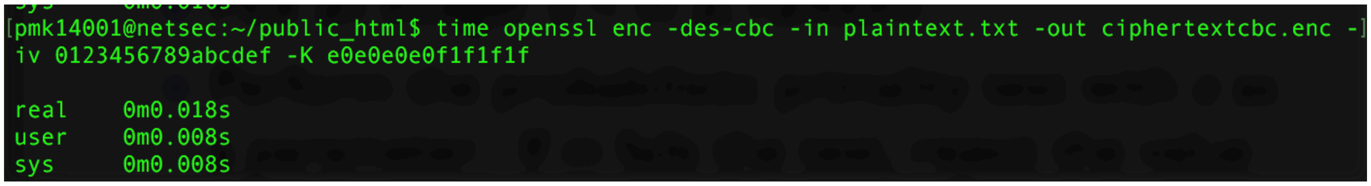
```
[pmk14001@netsec:~/public_html$ time openssl rsautl -encrypt -pubin -inkey pubkey.pem -in plaintext.]
 txt -out rsaencrypted.txt

 real    0m0.018s
 user    0m0.004s
 sys     0m0.012s
[pmk14001@netsec:~/public_html$ time openssl rsautl -decrypt -inkey private_key.priv -in rsaencrypte]
 d.txt -out rsadecrypted.txt

 real    0m0.021s
 user    0m0.004s
 sys     0m0.016s
```

3v. Run the DES encryption in CBC mode. What is the time for this operation?

Using the time command in UNIX, I was able to run the DES encryption in cbc mode again and got 0m0.008s for user time. My results can be shown below:



```
[pmk14001@netsec:~/public_html$ time openssl enc -des-cbc -in plaintext.txt -out ciphertextcbc.enc -]
 iv 0123456789abcdef -K e0e0e0e0f1f1f1f1

 real    0m0.018s
 user    0m0.008s
 sys     0m0.008s
```

[redacted]

4. In the last part of this assignment we'll look at certificates.

4a. Open either browser or operating system certificate store.

4i. What certificate store are you using?

Since I am a MacBook user, when going to my certificates on Chrome it opened up the certificates in the program Keychain access. Within there, the main certificate seems to be called "AddTrust External CA Root" which is a root certificate authority.

4ii. Find the COMODO Certification Authority Certificate. What country is this certificate from?

When looking at the COMODO Certificate, I saw the country to be GB (Great Britain) with the State/Province as Greater Manchester.

4iv. What size RSA key is being used for this certificate?

The size of the RSA key for this certificate is 2048 bits.

4v. How long is this certificate valid?

It shows that this certificate expires on Saturday, May 30, 2020 at 6:48:38 AM Eastern Daylight Time.

4vi. How many certificates authorities are in your trusted store?

Currently, I have 10 certificate authorities in my trusted login store but my overall system roots have 175 certificates.

4b. Connect to the websites google.com, chase.com and whitehouse.gov. What is the public key algorithm in each certificate?

Google- Public Key Algorithm = RSA Encryption (1.2.840.113549.1.1.1)

Chase- Public Key Algorithm = RSA Encryption (1.2.840.113549.1.1.1)

Whitehouse- Public Key Algorithm = RSA Encryption (1.2.840.113549.1.1.1)