CP400S – LAB REPORTS 04, 05, 06

David Moreno-Bautista
151925580
more5580@mylaurier.ca

Lab 04 – Private Key Cryptography

This lab introduces private-key cryptography algorithms. More specifically, it demonstrates how private-key algorithms work and how encryption modes work.

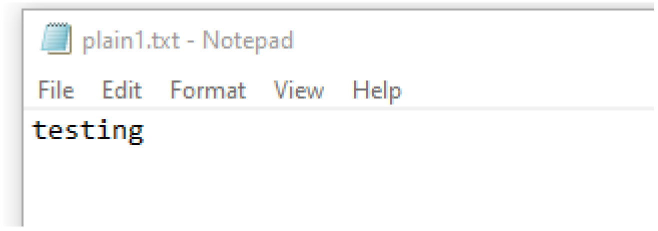
Preparation

- Install OpenSSL from: https://wiki.openssl.org/index.php/Binaries
- Add “C:\OpenSSL-Win64\bin” to Windows variable “Path” if it is not there.
- Install a binary file editor. Some examples are HxD, Neo, etc.

Observations

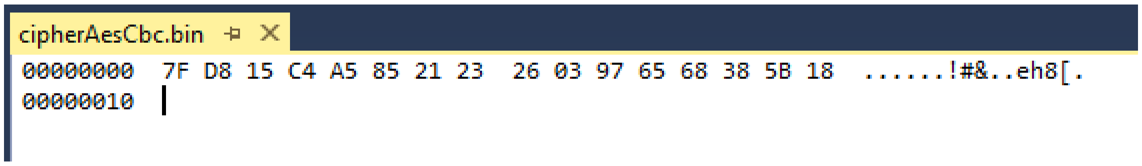
TASK 1

- Starting out with a plaintext file that looks like this:

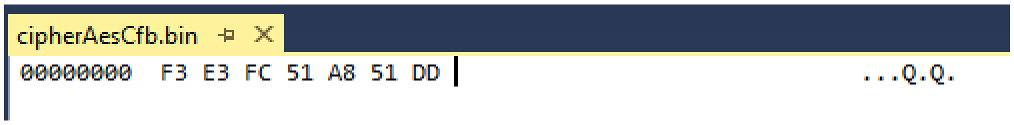


```
testing
```

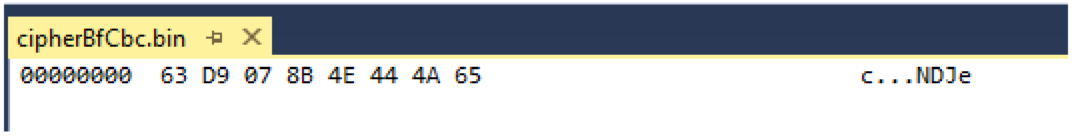
- Using openssl to encrypt the file with AES-128 in CBC mode and CFB mode as well as BF encryption in CBC mode gives the following three files:



```
00000000  7F D8 15 C4 A5 85 21 23  26 03 97 65 68 38 5B 18  .......!#&..eh8[.
00000010
```



```
00000000  F3 E3 FC 51 A8 51 DD                               ...Q.Q.
```



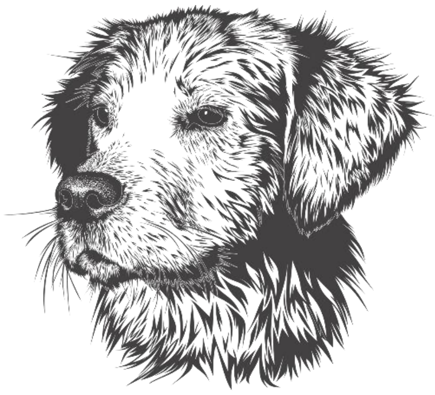
```
00000000  63 D9 07 8B 4E 44 4A 65                            c...NDJe
```

- The following command would encrypt the plaintext file using BF encryption in CBC mode: openssl.exe enc -bf-cbc -e -in plain1.txt -out cipherBfCbc.bin -K 00112233445566778889aabbccddeeff -iv 0102030405060708
- The following command would decrypt the cyphertext file using BF decryption in CBC mode: openssl.exe enc -bf-cbc -d -in cipherBfCbc.bin -out plainBfCbc.txt -K 00112233445566778889aabbccddeeff -iv 0102030405060708

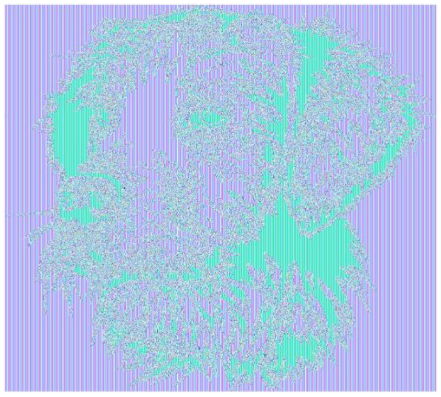
- The same idea is implemented with all other encryption modes, and they all return the original text file when the decryption is performed.

TASK 2

- Starting out with a bmp image file that looks like this:



- Using openssl to encrypt the file in ECB mode, and then placing back the 36 byte header to make the image viewable gives the following encrypted image file:



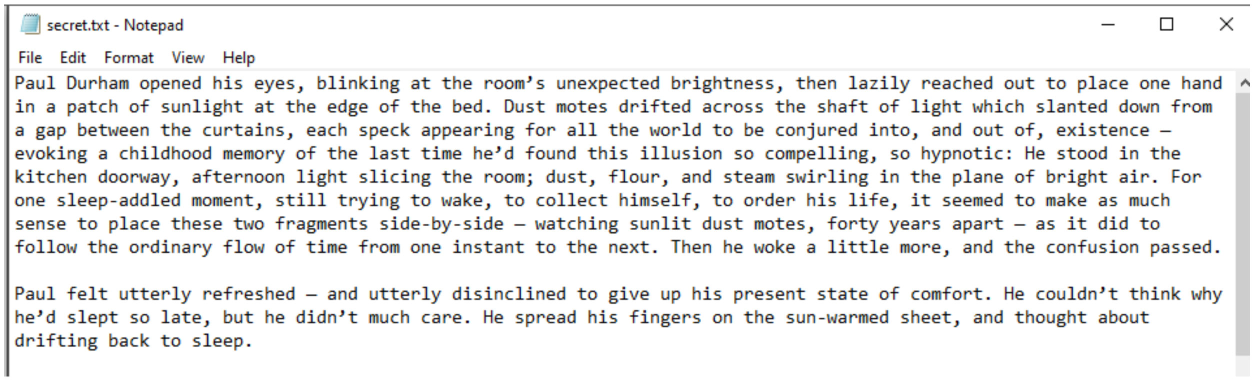
- Using openssl to encrypt the file in CBC mode, and then replacing back the 36 byte header to make the image viewable gives the following encrypted image file:

- The previous images show that when using ECB mode, a lot of information from the original picture can be derived, since you can clearly see the outline of a dog. This is due to the independent block encryption done in ECB, which maintains all of the statistical patterns found in the plaintext.

- On the other hand, the CBC encrypted image retains no useful information from the plaintext because it is a chain encryption. Thus, all of the information is scrambled much more thoroughly in CBC, hiding all of the significant information found in the plaintext.

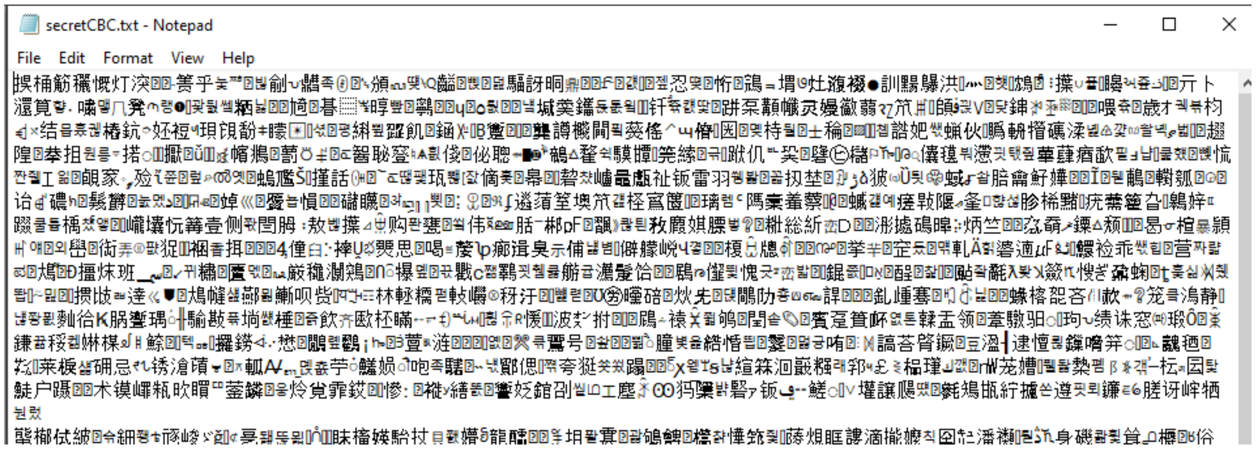
TASK 3

- Starting out with a plaintext file that looks like this:

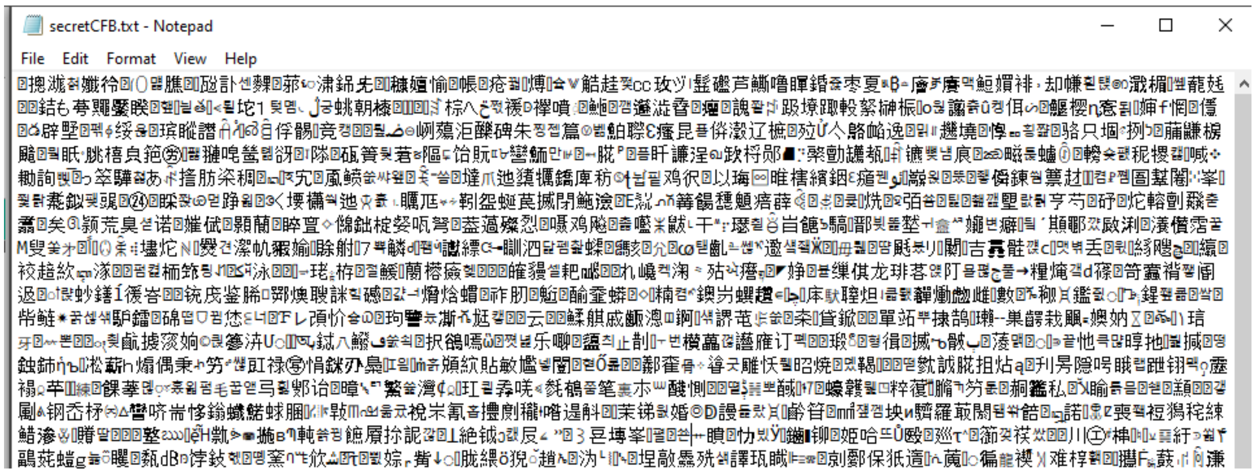


```
Paul Durham opened his eyes, blinking at the room's unexpected brightness, then lazily reached out to place one hand
in a patch of sunlight at the edge of the bed. Dust motes drifted across the shaft of light which slanted down from
a gap between the curtains, each speck appearing for all the world to be conjured into, and out of, existence –
evoking a childhood memory of the last time he'd found this illusion so compelling, so hypnotic: He stood in the
kitchen doorway, afternoon light slicing the room; dust, flour, and steam swirling in the plane of bright air. For
one sleep-addled moment, still trying to wake, to collect himself, to order his life, it seemed to make as much
sense to place these two fragments side-by-side – watching sunlit dust motes, forty years apart – as it did to
follow the ordinary flow of time from one instant to the next. Then he woke a little more, and the confusion passed.

Paul felt utterly refreshed – and utterly disinclined to give up his present state of comfort. He couldn't think why
he'd slept so late, but he didn't much care. He spread his fingers on the sun-warmed sheet, and thought about
drifting back to sleep.
```

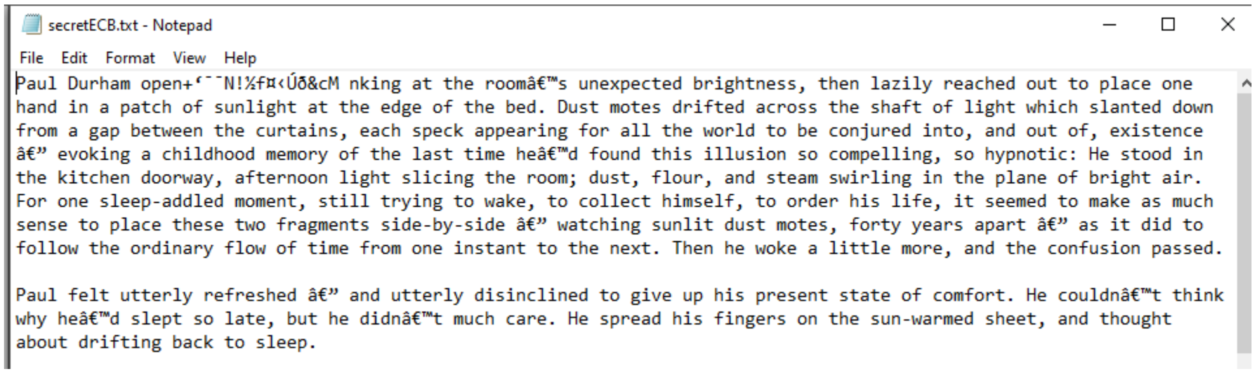
- Openssl is used to encrypt the file with AES-128 in CBC mode, CFB mode, and ECB mode. Then, the 30th byte is replaced with the number 99 in all three encrypted files to simulate a corruption in the files. Lastly, the corrupted files are decrypted accordingly to provide the following three files:



[illegible]



[illegible]



```
Paul Durham open+‘¯¯N!½f¤‹Úõ&cM nking at the roomâ€™s unexpected brightness, then lazily reached out to place one
hand in a patch of sunlight at the edge of the bed. Dust motes drifted across the shaft of light which slanted down
from a gap between the curtains, each speck appearing for all the world to be conjured into, and out of, existence
â€” evoking a childhood memory of the last time heâ€™d found this illusion so compelling, so hypnotic: He stood in
the kitchen doorway, afternoon light slicing the room; dust, flour, and steam swirling in the plane of bright air.
For one sleep-addled moment, still trying to wake, to collect himself, to order his life, it seemed to make as much
sense to place these two fragments side-by-side â€” watching sunlit dust motes, forty years apart â€” as it did to
follow the ordinary flow of time from one instant to the next. Then he woke a little more, and the confusion passed.

Paul felt utterly refreshed â€” and utterly disinclined to give up his present state of comfort. He couldnâ€™t think
why heâ€™d slept so late, but he didnâ€™t much care. He spread his fingers on the sun-warmed sheet, and thought
about drifting back to sleep.
```

- When the file was encrypted using CBC or CFB modes, none of the information can be recovered after the corruption. They are both very similar chain encryptions, so once corruption happens at one point, it spreads down the chain.
- On the other hand, when the file was encrypted using ECB, most of the information is recovered. There are only small sections where the characters don’t match, which correspond to the blocks affected. Since the corruption doesn’t spread from those blocks to the rest, a lot of the information is left untouched.

Reflections

- This lab helped me realize the importance of semantic security when it comes to encryption. Semantic security is defined as follows: “An adversary is allowed to choose between two plaintexts, m0 and m1, and he receives an encryption of either one of the plaintexts. An encryption scheme is semantically secure, if an adversary cannot guess with better probability than 1/2 whether the given ciphertext is an encryption of message m0 or m1” (Sako, 2011).
- As the above definition clearly outlines, the most important aspect of an encryption scheme is the reduction of any correlations between the plaintext and the ciphertext. The first task in this lab introduced me to the many variations of encryption schemes. Then, the second task helped me realize that the chained implementations of CBC and CFB are much more secure than ECB. The scrambling of the plaintext done in the latter encryption schemes help to define encryption schemes that most closely adhere to the semantic security standard.
- Lastly, the third task helped demonstrate what happens to encryption in the context of corrupted files. Even though corruption does not affect ECB encryption nearly as much as it does CBC or CFB, it comes to show how less secure it truly is. It is better to have to resend a file that got corrupted during encryption, which is not expected to happen so often to be a major problem, than to use an encryption scheme that can so easily be analyzed in such ways that defies the purpose of the encryption in the first place.

References

Sako K. (2011) Semantic Security. In: van Tilborg H.C.A., Jajodia S. (eds) Encyclopedia of Cryptography and Security. Springer, Boston, MA. https://doi.org/10.1007/978-1-4419-5906-5

Lab 05 – One-Way Hash Function and HMAC

This lab familiarizes students with one-way hash functions and Message Authentication Code (MAC). Students will learn to use tools to generate one-way hash values and HMAC for a given message.

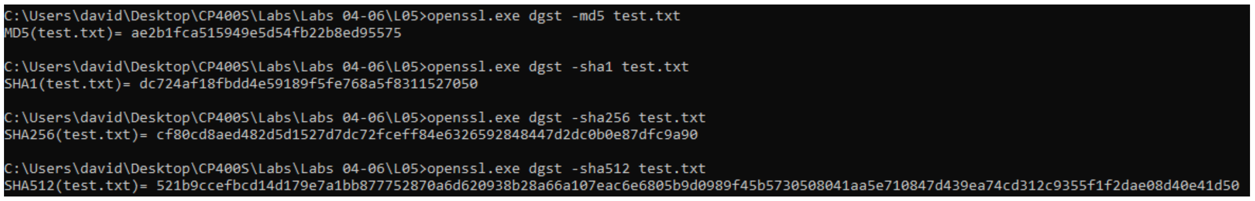
Preparation

- Install OpenSSL from: https://wiki.openssl.org/index.php/Binaries
- Add “C:\OpenSSL-Win64\bin” to Windows variable “Path” if it is not there.
- Install a binary file editor. Some examples are HxD, Neo, etc.

Observations

TASK 1

- Starting out with a 7 byte file called test.txt, it is hashed with md5, sha1, sha256, and sha256 functions using openssl as follows:



```
C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -md5 test.txt
MD5(test.txt)= ae2b1fca515949e5d54fb22b8ed95575

C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -sha1 test.txt
SHA1(test.txt)= dc724af18fbdd4e59189f5fe768a5f8311527050

C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -sha256 test.txt
SHA256(test.txt)= cf80cd8aed482d5d1527d7dc72fceff84e6326592848447d2dc0b0e87dfc9a90

C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -sha512 test.txt
SHA512(test.txt)= 521b9ccefbcd14d179e7a1bb877752870a6d620938b28a66a107eac6e6805b9d0989f45b5730508041aa5e710847d439ea74cd312c9355f1f2dae08d40e41d50
```

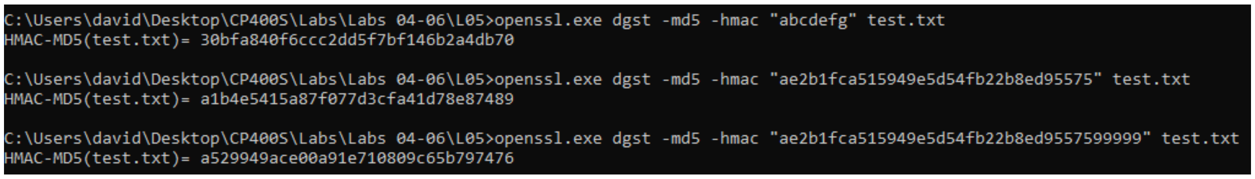
- The number of bytes produced from each hash function are as follows:

Hash algorithm	Output size (bytes/bits)	Note
MD5	16/128	4 bits * 32 chars = 128 bits = 16 bytes
SHA-1	20/160	4 bits * 40 chars = 160 bits = 20 bytes
SHA-256	32/256	4 bits * 64 chars = 256 bits = 32 bytes
SHA-512	64/512	4 bits * 128 chars = 512 bits = 64 bytes

- Therefore, each hash function works as expected.

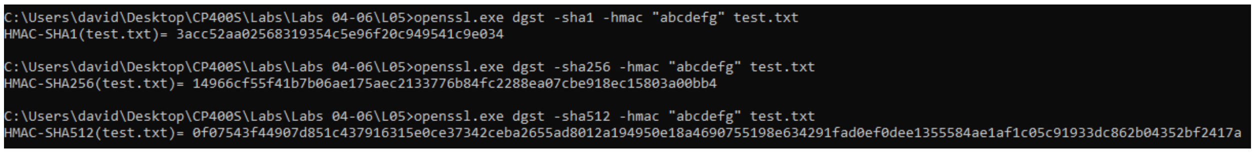
TASK 2

- Using the same 7 byte file called test.txt, a keyed hash is generated for it with an HMAC-MD5 function and keys of various sizes using openssl as follows:



```
C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -md5 -hmac "abcdefg" test.txt
HMAC-MD5(test.txt)= 30bfa840f6ccc2dd5f7bf146b2a4db70

C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -md5 -hmac "ae2b1fca515949e5d54fb22b8ed95575" test.txt
HMAC-MD5(test.txt)= a1b4e5415a87f077d3cfa41d78e87489

C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -md5 -hmac "ae2b1fca515949e5d54fb22b8ed9557599999" test.txt
HMAC-MD5(test.txt)= a529949ace00a91e710809c65b797476
```

- The first key is shorter than the output text, the second is the same size, and the last one is longer to show that all sizes of keys produce a valid result.
- Furthermore, a keyed hash is generated for the same file with HMAC-SHA1, HMAC-SHA256, and HMAC-SHA512 functions using openssl as follows:

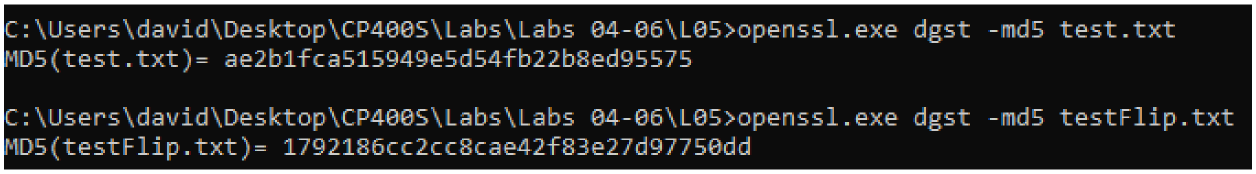


```
C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -sha1 -hmac "abcdefg" test.txt
HMAC-SHA1(test.txt)= 3acc52aa02568319354c5e96f20c949541c9e034

C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -sha256 -hmac "abcdefg" test.txt
HMAC-SHA256(test.txt)= 14966cf55f41b7b06ae175aec2133776b84fc2288ea07cbe918ec15803a00bb4

C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -sha512 -hmac "abcdefg" test.txt
HMAC-SHA512(test.txt)= 0f07543f44907d851c437916315e0ce37342ceba2655ad8012a194950e18a4690755198e634291fad0ef0dee1355584ae1af1c05c91933dc862b04352bf2417a
```

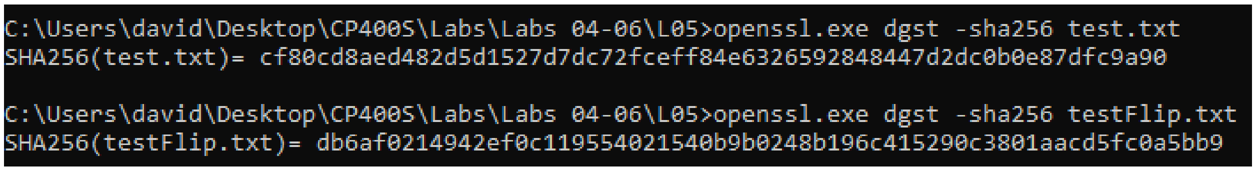
- As such, all keyed hash functions generate a hash with the appropriate size regardless of size length.
- In summary, a key with a fixed size is not necessary for HMAC. However, the ideal size is one that is equal to the block size (the size of the output text), because a smaller key size would be padded with zeroes to match the block size, which would be less secure in theory. Furthermore, a longer key would be rehashed to create a key of matching length to the block size, which could be avoided by simply providing a true random key of the same size as the block size.

TASK 3

- Using the same 7 byte file called test.txt, a hash is generated with MD5 and SHA256 functions using openssl. Then, one of the bytes is flipped using a binary editor and a new hash is generated using both functions again for the modified file. These are the results:



```
C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -md5 test.txt
MD5(test.txt)= ae2b1fca515949e5d54fb22b8ed95575

C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -md5 testFlip.txt
MD5(testFlip.txt)= 1792186cc2cc8cae42f83e27d97750dd
```



```
C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -sha256 test.txt
SHA256(test.txt)= cf80cd8aed482d5d1527d7dc72fceff84e6326592848447d2dc0b0e87dfc9a90

C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L05>openssl.exe dgst -sha256 testFlip.txt
SHA256(testFlip.txt)= db6af0214942ef0c119554021540b9b0248b196c415290c3801aacd5fc0a5bb9
```

- Despite being different only by one byte, there are no similarities between the original file and the modified file using either function. Hashing algorithms are thoroughly tested to minimize the amount of collisions generated and would not have been implemented if any information about the original file could be gathered by looking at the hash. In summary, the purpose of the hash functions is to create a unique representation of the original file with no statistical similarities to provide the best security.

Reflections

- This lab helped to expand my understanding of semantic security further as a continuation of the previous lab. The first task introduced some of the many hashing algorithms that are available. Then, the second task expanded on this by showing their use in the context of keyed message authentication codes. Lastly, the third task solidified the connection between hashing functions and semantic security.
- After completing the third task, this made me realize the significance of the fact that MD5 and SHA-1 “have been proven to be insecure [and] subject to collision attacks” (Velvindron et al., 2019). These older hashing functions are being phased out, which comes as a surprise from how they appear to be completely random in the generation of the hash. Just by comparing the hashes of almost identical files, it is hard to believe that the MD5 function can actually generate statistically similar hashes when used in some well-crafted test cases.
- The world of cryptography is constantly evolving and moving at a much faster pace in the present day. Algorithms are constantly tested, refined, and whole new versions are developed as new knowledge is acquired.

References

Velvindron L., Moriarty K., Ghedini A. (2019, November 30). Deprecating MD5 and SHA-1 signature hashes in TLS 1.2. Internet Engineering Task Force. Retrieved March 26, 2020, from https://tools.ietf.org/id/draft-lvelvindron-tls-md5-sha1-deprecate-05.html

Lab 06 – Public-Key Cryptography and Digital Signature Algorithm

This lab familiarizes students with public-key encryption and digital signature algorithms. Students will be able to use openssl tools and to exchange messages secretly.

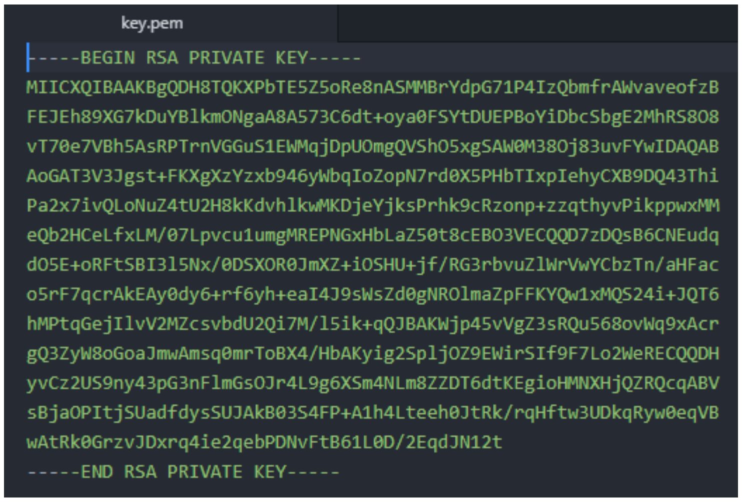
Preparation

- Install OpenSSL from: https://wiki.openssl.org/index.php/Binaries
- Add “C:\OpenSSL-Win64\bin” to Windows variable “Path” if it is not there.

Observations

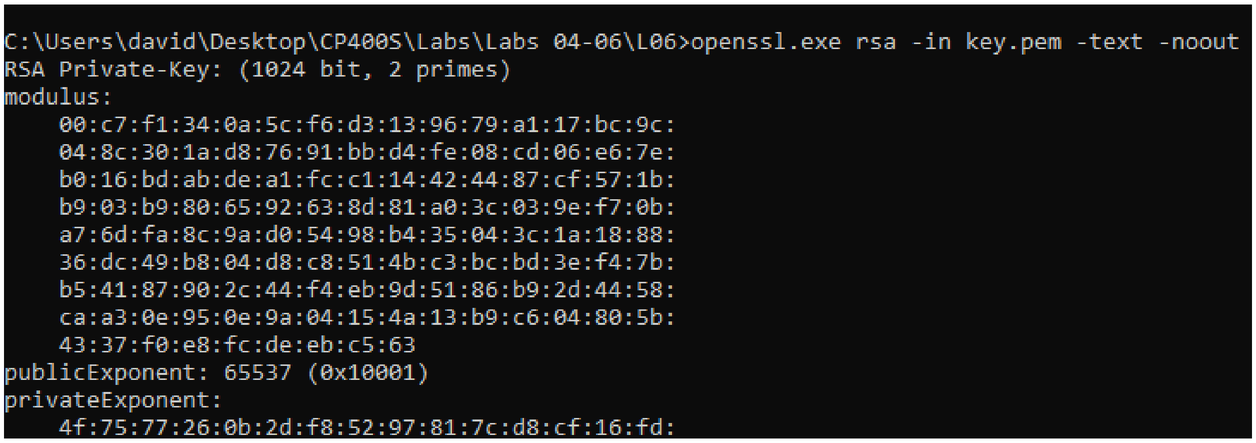
TASK 1

- Using openssl, we generate a private key using the RSA algorithm:



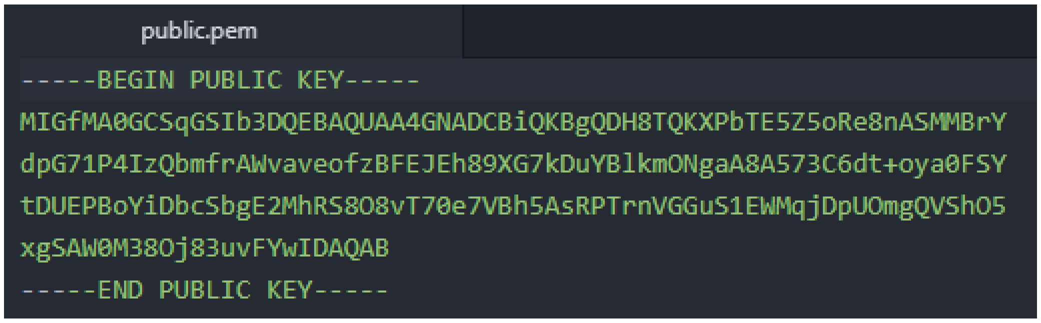
```
-----BEGIN RSA PRIVATE KEY-----
MIICXQIBAAKBgQDH8TQKXPbTE5Z5oRe8nASMMBrYdpG71P4IzQbmfrAWvaveofzB
FEJEh89XG7kDuYBlkmONgaA8A573C6dt+oya0FSYtDUEPBoYiDbcSbgE2MhRS8O8
vT70e7VBh5AsRPTrnVGGuS1EWMqjDpUOmgQVShO5xgSAW0M38Oj83uvFYwIDAQAB
AoGAT3V3Jgst+FKXgXzYzxb946yWbqIoZopN7rd0X5PHbTIxpIehyCXB9DQ43Thi
Pa2x7ivQLoNuZ4tU2H8kKdvhlkwMKDjeYjksPrhk9cRzonp+zzqthyvPikppwxMM
eQb2HCeLfxLM/07Lpvcu1umgMREPNGxHbLaZ50t8cEBO3VECQQD7zDQsB6CNEudq
dO5E+oRFtSBI3l5Nx/0DSXOR0JmXZ+iOSHU+jf/RG3rbvuZlWrVwYCbzTn/aHFac
o5rF7qcrAkEAy0dy6+rf6yh+eaI4J9sWsZd0gNROlmaZpFFKYQw1xMQS24i+JQT6
hMPtqGejIlvV2MZcsvbdU2Qi7M/l5ik+qQJBAKWjp45vVgZ3sRQu568ovWq9xAcr
gQ3ZyW8oGoaJmwAmsq0mrToBX4/HbAKyig2SpljOZ9EWirSIf9F7Lo2WeRECQQDH
yvCz2US9ny43pG3nFlmGsOJr4L9g6XSm4NLm8ZZDT6dtKEgioHMNXHjQZRQcqABV
sBjaOPItjSUadfdysSUJAkB03S4FP+A1h4Lteeh0JtRk/rqHftw3UDkqRyw0eqVB
wAtRk0GrzvJDxrq4ie2qebPDNvFtB61L0D/2EqdJN12t
-----END RSA PRIVATE KEY-----
```

- This was generated with the command: openssl.exe genrsa -out key.pem 1024
- Openssl can also be used to display the details of the RSA key pair, such as the modulus:



```
C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L06>openssl.exe rsa -in key.pem -text -noout
RSA Private-Key: (1024 bit, 2 primes)
modulus:
    00:c7:f1:34:0a:5c:f6:d3:13:96:79:a1:17:bc:9c:
    04:8c:30:1a:d8:76:91:bb:d4:fe:08:cd:06:e6:7e:
    b0:16:bd:ab:de:a1:fc:c1:14:42:44:87:cf:57:1b:
    b9:03:b9:80:65:92:63:8d:81:a0:3c:03:9e:f7:0b:
    a7:6d:fa:8c:9a:d0:54:98:b4:35:04:3c:1a:18:88:
    36:dc:49:b8:04:d8:c8:51:4b:c3:bc:bd:3e:f4:7b:
    b5:41:87:90:2c:44:f4:eb:9d:51:86:b9:2d:44:58:
    ca:a3:0e:95:0e:9a:04:15:4a:13:b9:c6:04:80:5b:
    43:37:f0:e8:fc:de:eb:c5:63
publicExponent: 65537 (0x10001)
privateExponent:
    4f:75:77:26:0b:2d:f8:52:97:81:7c:d8:cf:16:fd:
```

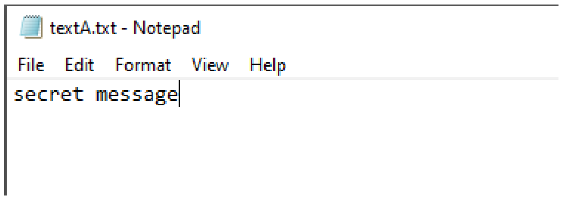
- Furthermore, we generate a public key from the private key information:



public.pem

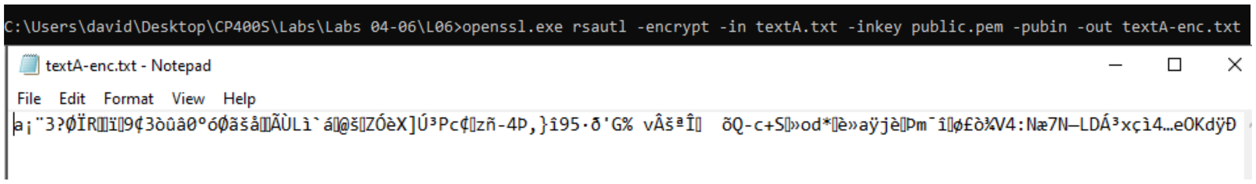
```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDH8TQKXPbTE5Z5oRe8nASMMBrY
dpG71P4IzQbmfrAWvaveofzBFEJEh89XG7kDuYBlkmONgaA8A573C6dt+oya0FSY
tDUEPBoYiDbcSbgE2MhRS8O8vT70e7VBh5AsRPTrnVGGuS1EWMqjDpUOmgQVShO5
xgSAW0M38Oj83uvFYwIDAQAB
-----END PUBLIC KEY-----
```

- This was generated with the command: openssl.exe rsa -in key.pem -pubout -out public.pem
- Now, we can begin encryption and decryption. We start out with the following plaintext file:



```
secret message
```

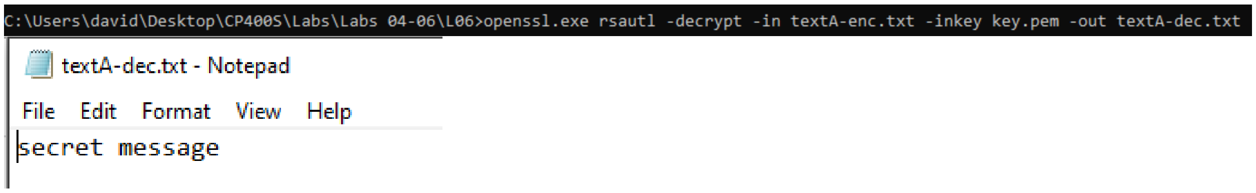
- Take Alice for example. She would encrypt this secret message using Bob's public key (the one we generated) and produce the following ciphertext:



```
C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L06>openssl.exe rsautl -encrypt -in textA.txt -inkey public.pem -pubin -out textA-enc.txt
```

```
a¡¨3?ØÏR▯ï▯9¢3òûâ0°óØãšå▯ÃÙLì`á▯@š▯ZÓèX]Ú³Pc¢▯zñ-4Þ,}î95·ð'G% vÂšªÎ▯  õQ-c+S▯»od*▯è»aÿjè▯Þm¯î▯ø£ò¾V4:Næ7N—LDÁ³xçì4…eOKdÿÐ
```

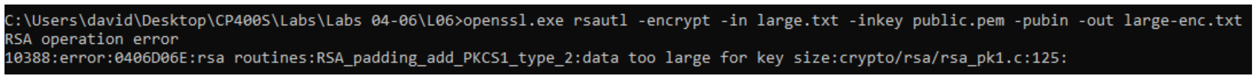
- Alice would send this message to Bob, and he would decrypt it using his private key (the one we generated at the beginning):



```
C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L06>openssl.exe rsautl -decrypt -in textA-enc.txt -inkey key.pem -out textA-dec.txt
```

```
secret message
```

- Our public key is public and anyone can use it to encrypt messages and send it to us securely since they can only be decrypted using our private key (which no one else knows).
- The only caveat with RSA is that large enough files cannot be encrypted:

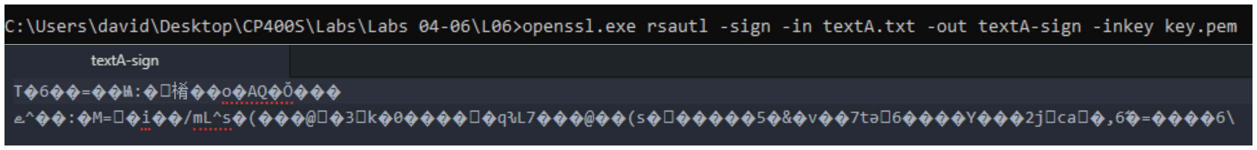


```
C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L06>openssl.exe rsautl -encrypt -in large.txt -inkey public.pem -pubin -out large-enc.txt
RSA operation error
10388:error:0406D06E:rsa routines:RSA_padding_add_PKCS1_type_2:data too large for key size:crypto/rsa/rsa_pk1.c:125:
```

- Since RSA encryption relies on modular math, the produced bytes from the file cannot be larger than the modulus used to generate our keys. There would be no way to distinguish two characters that reduce to the same number using our modulus.

TASK 2

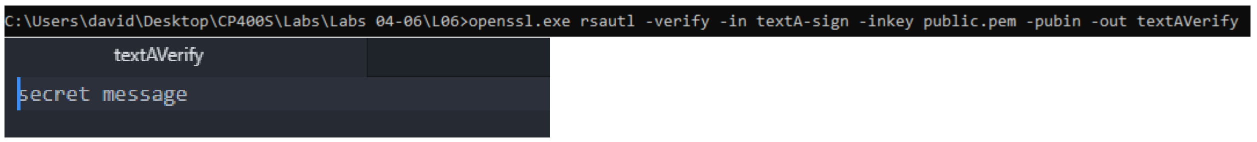
- Openssl can also be used for digital signatures. Using the same textA.txt secret message, we can sign the file using our private key:



```
C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L06>openssl.exe rsautl -sign -in textA.txt -out textA-sign -inkey key.pem
```

```
T�6��=��M:�□楕��o�AQ�Ŏ���
ഛ^��:�M=□�i��/mL^s�(���@□�3□k�0�����□�q꜒L7���@��(s�□������5�&�v��7tə□6����Y���2j□ca□�,6�=����6\
```

- After sending this file to someone else, they can use our public key to verify that it came from us:



```
C:\Users\david\Desktop\CP400S\Labs\Labs 04-06\L06>openssl.exe rsautl -verify -in textA-sign -inkey public.pem -pubin -out textAVerify
```

```
secret message
```

- If they were to use their own public key to decrypt our signed (encrypted) file, they would not be able to produce a valid decryption. In order to produce a valid decryption, the file would've had to have been encrypted with their own private key.
- The idea is that only we have our own private key, so only we could have encrypted something using our private key. The only way to decrypt our encrypted file is by using our public key, which is available for everyone to verify files encrypted by us. If someone else pretended to be us, decrypting their file with our public key would not produce a valid plaintext file.

TASK 3

- The RSA key generation algorithm is as follows:
- Choose two different prime numbers p and q.
  - E.g. p = 17, q = 11
- Calculate n = p * q.
  - E.g. n = 17 * 11 = 187
- Calculate t(n) = (p-1)(q-1)
  - E.g. t(n) = 16 * 10 = 160
- Choose e such that e is relatively prime to t(n) and less than t(n)
  - E.g. e = 7
- Calculate d such that d * e mod t(n) = 1 and d < t(n)
  - E.g. d = 23, since 23 * 7 = 161 = (1 * 160) + 1
- The public key is now {e, n} and the private key is {d, n}
  - E.g. public key = {7, 187} and private key is {23, 187}
- Now for a plaintext block M, encryption can proceed as Ciphertext = M^e (mod n)
  - E.g. Ciphertext = M^7 (mod 187)

- Similarly decryption for a ciphertext block C can proceed as Plaintext = C^d (mod n)
    - E.g. Plaintext = C^{23} (mod 187)

Reflections

- This lab helped demonstrate the efficiency of public key encryption. The first task showed how private and public keys are generated and their relationship to one another. Then, the actual process of encryption and decryption is walked through in this same task. Afterwards, the second task goes into the process of digital signatures in a similar fashion. Lastly, the third task goes into detail of how keys are generated mathematically.
- Modular math is very convenient and elegant in how it makes for a very simple, yet effective way of managing encryption and decryption. It allows for a public key that everyone can have the access to and an encryption algorithm that everyone can have knowledge of, all of which has zero negative effects on security.
- Looking for any weaknesses in RSA, I ran into the idea that if "an attacker can convince a key holder to sign an unformatted encrypted message using the same key then she gets the original" (Ireland, 2019). This is the case because encryption is done using one's public key, while signing is done using one's private key. Thus, the simplicity of RSA can sometimes be a drawback. On a positive note, this comes to show that the weakness is not in the technology itself but on the creative workarounds that people can come up with.

References

Ireland, David. (2019, December 21). RSA Algorithm. DI Management Services. Retrieved March 26, 2020, from https://www.di-mgt.com.au/rsa_alg.html