Timothy Jelinek

CIS313, Cryptography
Module 6 Lab – RSA and Diffie-Hellman Asymmetric Encryption and Signatures

In this sixth module, we will examine the use of asymmetric cryptographic algorithms for various purposes. Asymmetric algorithms are rarely used for bulk encryption. Rather they are used to for key exchanges over a public network, encryption of a bulk encrypting symmetric key or for digital signatures. This is because encryption using an asymmetric algorithm is much slower than a symmetric algorithm. The benefits are that you encrypt and decrypt with separate keys and gain authentication. In this lab you will generate public and private keys, sign a file, encrypt the file with AES and then encrypt the symmetric key for storage, and use Diffie-Hellman to generate a shared symmetric key.

You will be required to submit the following graded items as part of this lab:

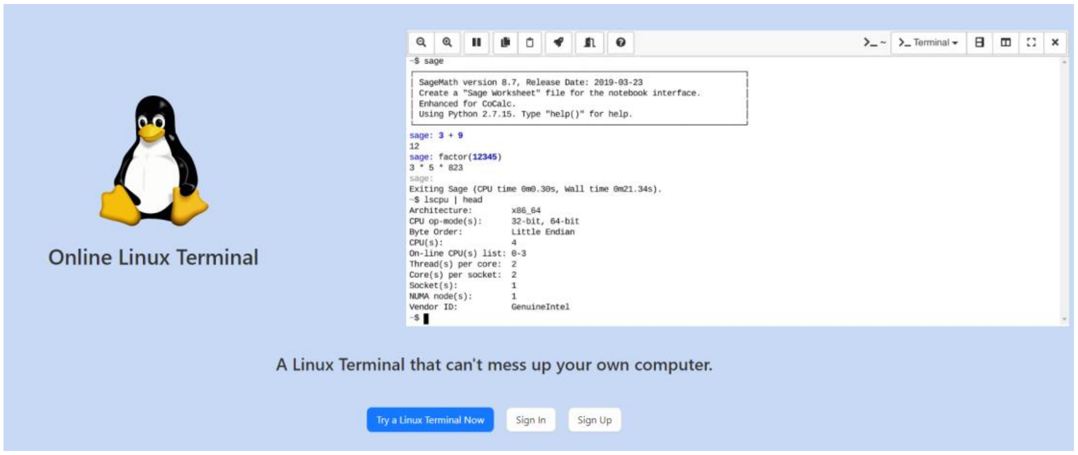
- A screenshot and text of your public and private keys
- A screenshot of a valid signature
- A screenshot of an invalid signature
- A screenshot of the decrypted 2b.txt file
- A screenshot of the Diffie-Hellman prime n and generator g
- A screenshot of the Diffie-Hellman shared secret
- A screenshot of the KDF shared AES key

Lab 1 – RSA Key Generation and Digital Signatures

In this lab we will explore salted and unsalted password hashing, including illustrating the danger of unsalted password hashes.

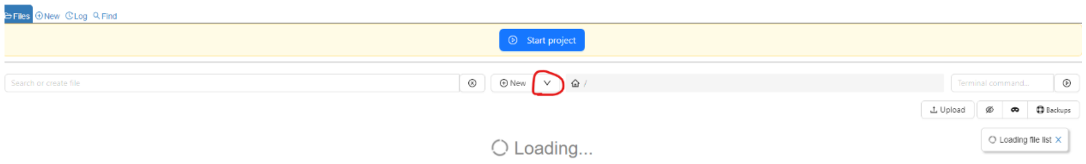
If using CoCalc:

- Open your web browser and surf to https://cocalc.com/doc/terminal.html. (Note: If you have access to a Linux OS you can also use that Linux terminal/command line to do this lab.)
- Click on the Blue 'Try a Linux Terminal Now' button.



```
~$ sage
┌────────────────────────────────────────────────────────────────────┐
│ SageMath version 8.7, Release Date: 2019-03-23                     │
│ Create a "Sage Worksheet" file for the notebook interface.         │
│ Enhanced for CoCalc.                                               │
│ Using Python 2.7.15. Type "help()" for help.                       │
└────────────────────────────────────────────────────────────────────┘
sage: 3 + 9
12
sage: factor(12345)
3 * 5 * 823
sage:
Exiting Sage (CPU time 0m0.30s, Wall time 0m21.34s).
~$ lscpu | head
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
CPU(s):              4
On-line CPU(s) list: 0-3
Thread(s) per core:  2
Core(s) per socket:  2
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
~$
```

- Click the blue “Use CoCalc Anonymously” button
- Select ‘Projects’ from the menu bar under the ‘Signed in as’ title
- Click the ‘Create Project’ button. You can leave the default project title as is.
- Click the blue ‘Start Project’ button.
- Click on the down arrow to the right of “New” and select Terminal



- Accept defaults for name for Terminal file, then click ‘Create.’

If using Windows:

- If you are using the Cybersecurity Desktop, open the command line by typing cmd in the Type here to search box on the taskbar. The type the following command:

```
set PATH=%PATH%;C:\Program Files\OpenSSL-Win64\bin
```

[Animated Example]

1. Next type the following commands to generate an RSA public and private key pair and store them in a file. The same commands work for both Windows and Linux/CoCalc.

CoCalc:

```
openssl genrsa -out private.key 2048
openssl rsa -in private.key -out public.key -outform PEM -pubout
```

Windows:

```
openssl genrsa -out private.key 2048
openssl rsa -in private.key -out public.key -outform PEM -pubout
```

These commands does the following:

- openssl – This is the name of the command we are using. It has many cryptographic functions.
- genrsa – This tells OpenSSL that we will be generate an RSA private key
- -out - Store the result of the operation in a file
- 2048 - The number of bits in the key to be generated
- rsa - Tells OpenSSL that we will working with RSA keys
- -in - Read input necessary for the command (private key in this case)
- -outform - Specifies how to format the output
- -pubout - Output the public key from this operation

View the public or private keys in the following ways:

CoCalc:

```
cat private.key; cat public.key
```

Windows:

```
more private.key & more public.key
```

CoCalc Example:

Animated Example

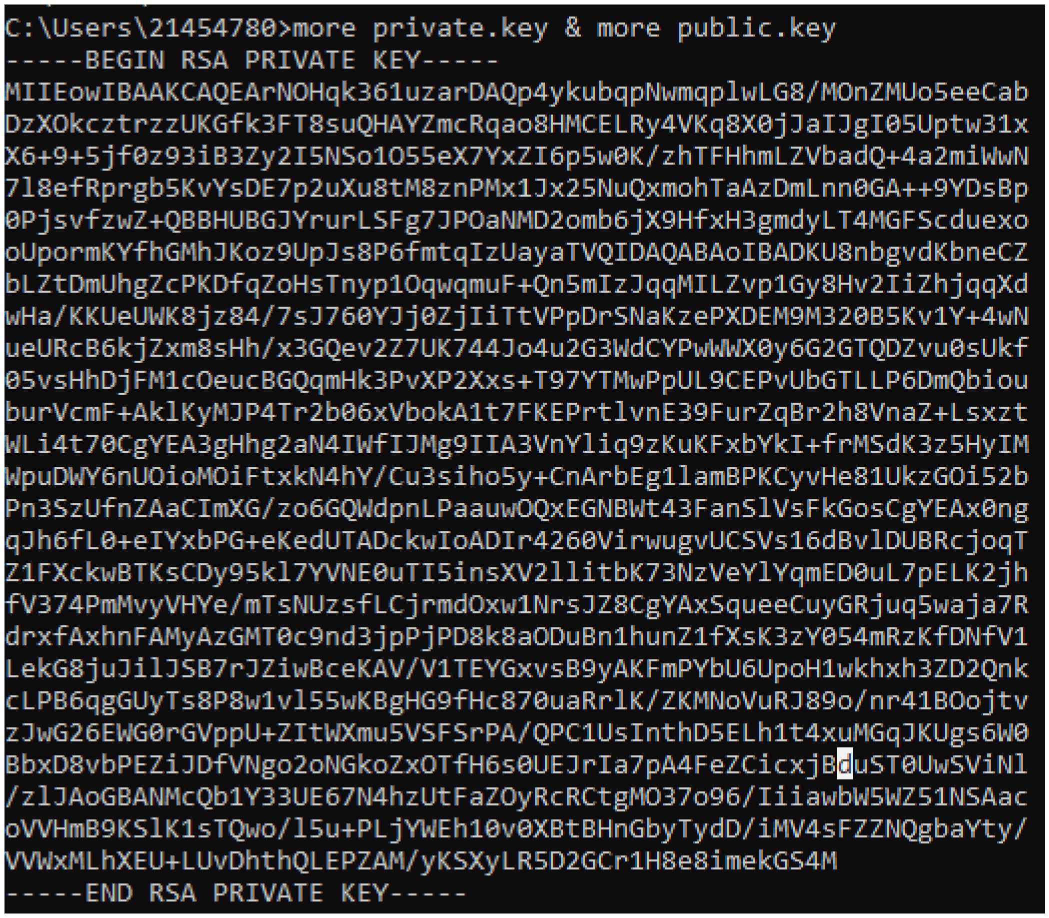
Windows Example:

Animated Example

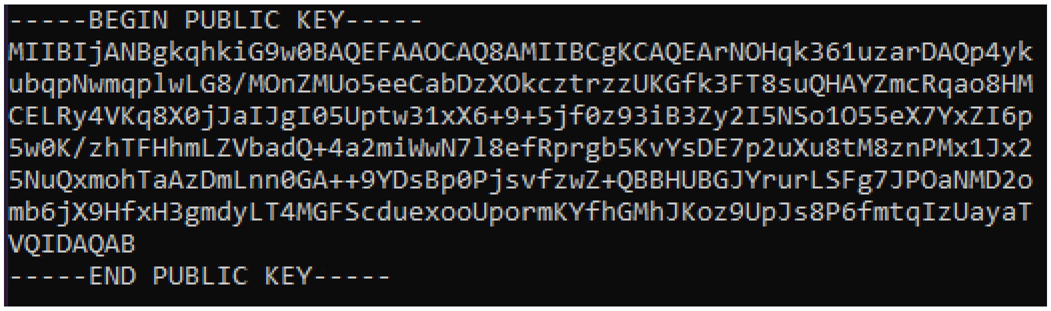
Record your public and private key below:

<u>Note: You are sharing your private key in this exercise because it is necessary for grading and you are not encrypting sensitive information with the key. Never, ever, ever, ever, EVER! share a private key with someone in the real world. Guard an RSA private key like you would your deepest, darkest secret.</u>

Private Key:



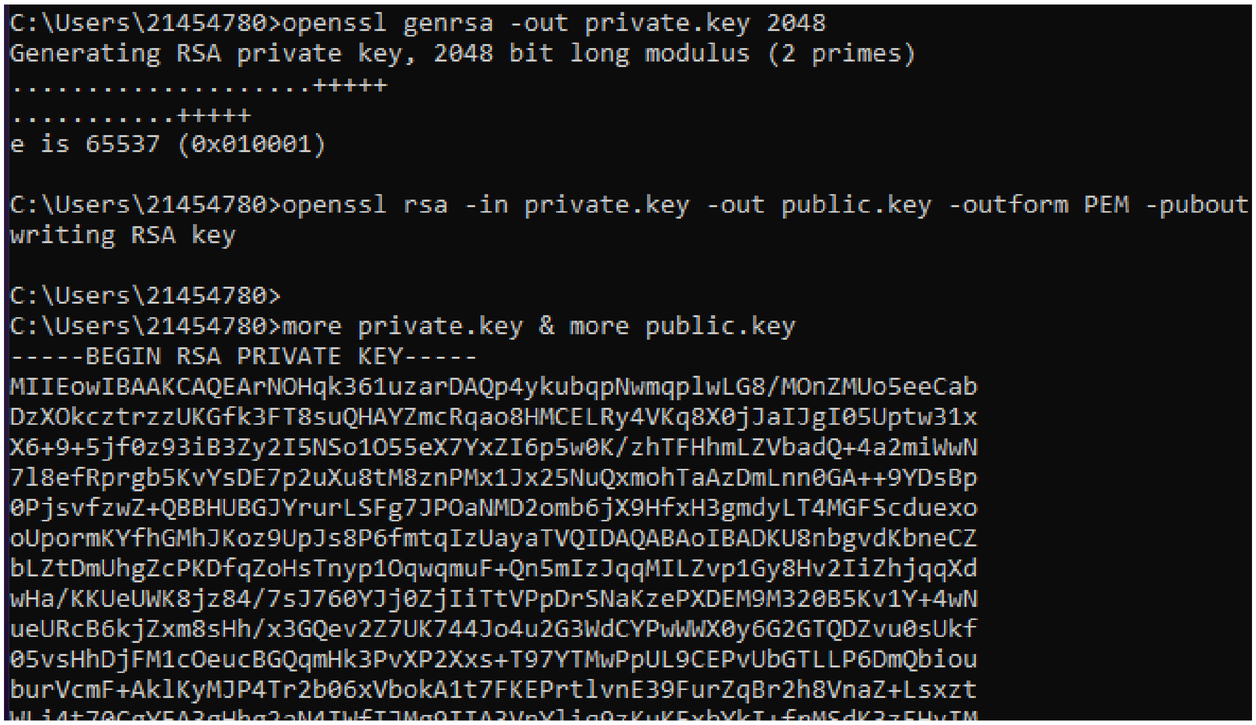
```
C:\Users\21454780>more private.key & more public.key
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEArNOHqk361uzarDAQp4ykubqpNwmqplwLG8/MOnZMUo5eeCab
DzXOkcztrzzUKGfk3FT8suQHAYZmcRqao8HMCELRy4VKq8X0jJaIJgI05Uptw31x
X6+9+5jf0z93iB3Zy2I5NSo1O55eX7YxZI6p5w0K/zhTFHhmLZVbadQ+4a2miWwN
7l8efRprgb5KvYsDE7p2uXu8tM8znPMx1Jx25NuQxmohTaAzDmLnn0GA++9YDsBp
0PjsvfzwZ+QBBHUBGJYrurLSFg7JPOaNMD2omb6jX9HfxH3gmdyLT4MGFScduexo
oUpormKYfhGMhJKoz9UpJs8P6fmtqIzUayaTVQIDAQABAoIBADKU8nbgvdKbneCZ
bLZtDmUhgZcPKDfqZoHsTnyp1OqwqmuF+Qn5mIzJqqMILZvp1Gy8Hv2IiZhjqqXd
wHa/KKUeUWK8jz84/7sJ760YJj0ZjIiTtVPpDrSNaKzePXDEM9M320B5Kv1Y+4wN
ueURcB6kjZxm8sHh/x3GQev2Z7UK744Jo4u2G3WdCYPwWWX0y6G2GTQDZvu0sUkf
05vsHhDjFM1cOeucBGQqmHk3PvXP2Xxs+T97YTMwPpUL9CEPvUbGTLLP6DmQbiou
burVcmF+AklKyMJP4Tr2b06xVbokA1t7FKEPrtlvnE39FurZqBr2h8VnaZ+Lsxzt
WLi4t70CgYEA3gHhg2aN4IWfIJMg9IIA3VnYliq9zKuKFxbYkI+frMSdK3z5HyIM
WpuDWY6nUOioMOiFtxkN4hY/Cu3siho5y+CnArbEg1lamBPKCyvHe81UkzGOi52b
Pn3SzUfnZAaCImXG/zo6GQWdpnLPaauwOQxEGNBWt43FanSlVsFkGosCgYEAx0ng
qJh6fL0+eIYxbPG+eKedUTADckwIoADIr4260VirwugvUCSVs16dBvlDUBRcjoqT
Z1FXckwBTKsCDy95kl7YVNE0uTI5insXV2llitbK73NzVeYlYqmED0uL7pELK2jh
fV374PmMvyVHYe/mTsNUzsfLCjrmdOxw1NrsJZ8CgYAxSqueeCuyGRjuq5waja7R
drxfAxhnFAMyAzGMT0c9nd3jpPjPD8k8aODuBn1hunZ1fXsK3zY054mRzKfDNfV1
LekG8juJilJSB7rJZiwBceKAV/V1TEYGxvsB9yAKFmPYbU6UpoH1wkhxh3ZD2Qnk
cLPB6qgGUyTs8P8w1vl55wKBgHG9fHc870uaRrlK/ZKMNoVuRJ89o/nr41BOojtv
zJwG26EWG0rGVppU+ZItWXmu5VSFSrPA/QPC1UsInthD5ELh1t4xuMGqJKUgs6W0
BbxD8vbPEZiJDfVNgo2oNGkoZxOTfH6s0UEJrIa7pA4FeZCicxjBduST0UwSViNl
/zlJAoGBANMcQb1Y33UE67N4hzUtFaZOyRcRCtgMO37o96/IiiawbW5WZ51NSAac
oVVHmB9KSlK1sTQwo/l5u+PLjYWEh10v0XBtBHnGbyTydD/iMV4sFZZNQgbaYty/
VVWxMLhXEU+LUvDhthQLEPZAM/yKSXyLR5D2GCr1H8e8imekGS4M
-----END RSA PRIVATE KEY-----
```

Public Key: 

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEArNOHqk361uzarDAQp4yk
ubqpNwmqplwLG8/MOnZMUo5eeCabDzXOkcztrzzUKGfk3FT8suQHAYZmcRqao8HM
CELRy4VKq8X0jJaIJgI05Uptw31xX6+9+5jf0z93iB3Zy2I5NSo1O55eX7YxZI6p
5w0K/zhTFHhmLZVbadQ+4a2miWwN7l8efRprgb5KvYsDE7p2uXu8tM8znPMx1Jx2
5NuQxmohTaAzDmLnn0GA++9YDsBp0PjsvfzwZ+QBBHUBGJYrurLSFg7JPOaNMD2o
mb6jX9HfxH3gmdyLT4MGFScduexooUpormKYfhGMhJKoz9UpJs8P6fmtqIzUayaT
VQIDAQAB
-----END PUBLIC KEY-----
```

What type of encoding is used to store these keys? asymmetric

Paste a screenshot of the commands that have been run:



```
C:\Users\21454780>openssl genrsa -out private.key 2048
Generating RSA private key, 2048 bit long modulus (2 primes)
....................+++++
...........+++++
e is 65537 (0x010001)

C:\Users\21454780>openssl rsa -in private.key -out public.key -outform PEM -pubout
writing RSA key

C:\Users\21454780>
C:\Users\21454780>more private.key & more public.key
-----BEGIN RSA PRIVATE KEY-----
MIIEowIBAAKCAQEArNOHqk361uzarDAQp4ykubqpNwmqplwLG8/MOnZMUo5eeCab
DzXOkcztrzzUKGfk3FT8suQHAYZmcRqao8HMCELRy4VKq8X0jJaIJgI05Uptw31x
X6+9+5jf0z93iB3Zy2I5NSo1O55eX7YxZI6p5w0K/zhTFHhmLZVbadQ+4a2miWwN
7l8efRprgb5KvYsDE7p2uXu8tM8znPMx1Jx25NuQxmohTaAzDmLnn0GA++9YDsBp
0PjsvfzwZ+QBBHUBGJYrurLSFg7JPOaNMD2omb6jX9HfxH3gmdyLT4MGFScduexo
oUpormKYfhGMhJKoz9UpJs8P6fmtqIzUayaTVQIDAQABAoIBADKU8nbgvdKbneCZ
bLZtDmUhgZcPKDfqZoHsTnyp1OqwqmuF+Qn5mIzJqqMILZvp1Gy8Hv2IiZhjqqXd
wHa/KKUeUWK8jz84/7sJ760YJj0ZjIiTtVPpDrSNaKzePXDEM9M320B5Kv1Y+4wN
ueURcB6kjZxm8sHh/x3GQev2Z7UK744Jo4u2G3WdCYPwWWX0y6G2GTQDZvu0sUkf
05vsHhDjFM1cOeucBGQqmHk3PvXP2Xxs+T97YTMwPpUL9CEPvUbGTLLP6DmQbiou
burVcmF+AklKyMJP4Tr2b06xVbokA1t7FKEPrtlvnE39FurZqBr2h8VnaZ+Lsxzt
```

2. Next create a file on the command line with the following content and commands.

File Content:

To be, or not to be, that is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of outrageous fortune,
Or to take arms against a sea of troubles
And by opposing end them. To die—to sleep,
No more; and by a sleep to say we end
The heart-ache and the thousand natural shocks
That flesh is heir to: 'tis a consummation
Devoutly to be wish'd. To die, to sleep;
To sleep, perchance to dream—ay, there's the rub:

For in that sleep of death what dreams may come,
When we have shuffled off this mortal coil,
Must give us pause—there's the respect
That makes calamity of so long life.
For who would bear the whips and scorns of time,
Th'oppressor's wrong, the proud man's contumely,
The pangs of dispriz'd love, the law's delay,
The insolence of office, and the spurns
That patient merit of th'unworthy takes,
When he himself might his quietus make
With a bare bodkin? Who would fardels bear,
To grunt and sweat under a weary life,
But that the dread of something after death,
The undiscovere'd country, from whose bourn
No traveller returns, puzzles the will,
And makes us rather bear those ills we have
Than fly to others that we know not of?
Thus conscience does make cowards of us all,
And thus the native hue of resolution
Is sicklied o'er with the pale cast of thought,
And enterprises of great pitch and moment
With this regard their currents turn awry
And lose the name of action.

If you are using CoCalc follow the instructions below to create the file to encrypt:

- Type vim 2b.txt at the command line ~$
- Type the letter i and look for the bottom of the command line to say --Insert--
- Right click the Text to Encrypt above and copy it
- Click on the CoCalc command line Window and press CTRL+v
- Press the ESC button and watch for the --Insert-- at the bottom to disappear
- Type :wq and then ENTER
- Type ls and ensure the file has been created and is named correctly

Animated Example

If you are using Windows follow the instructions below to create the file to encrypt:

- Right click the Desktop and then select New > Text Document
- Name the new text document tomorrow.txt
- Open the 2b.txt document
- Copy and paste the Test to Encrypt into the 2b.txt document
- Save and close the document
- If the command line windows isn't open type cmd in the Type Here to Search box on the taskbar and press ENTER

- Type cd Desktop
- Type dir to ensure the file has been correctly created

Animated Example

3. In the next step, we will create a digital signature of the file. You will use the private key you generated to create a signature. In this process, a hash of the text document is created and it is encrypted with the private key.

CoCalc:

```
openssl dgst -sha256 -sign private.key -out 2b.txt.sign 2b.txt
```

Windows:

```
openssl dgst -sha256 -sign private.key -out 2b.txt.sign 2b.txt
```

These commands do the following:

- openssl – This is the name of the command we are using. It has many cryptographic functions.
- dgst – This tells OpenSSL that we will be doing a hashing algorithm
- -sha256 - Use the SHA 256 hashing algorithm
- -out - Output the signature to the filename specified
- 2b.txt - The name of the file to be signed

CoCalc Example:

Animated Example

Windows Example:

Animated Example

4. Now we use SSL and the public key to verify the signature. Remember that something encrypted with the public key can only be decrypted with the private key and vice versa. If someone encrypts a hash with their private key and they are the only one with the private key, decrypting successfully with the public key provides authentication of that person and verifying the hash provides integrity.

CoCalc:

```
openssl dgst -sha256 -verify public.key -signature 2b.txt.sign 2b.txt
```

Windows:

```
openssl dgst -sha256 -verify public.key -signature 2b.txt.sign 2b.txt
```

These commands do the following:

- openssl – This is the name of the command we are using. It has many cryptographic functions.
- dgst – This tells OpenSSL that we will be doing a hashing algorithm
- -sha256 - Use the SHA 256 hashing algorithm
- -verify - Use the following public key to verify the signature
- -signature - The stored binary signature to use
- 2b.txt - The name of the file to check against the signature

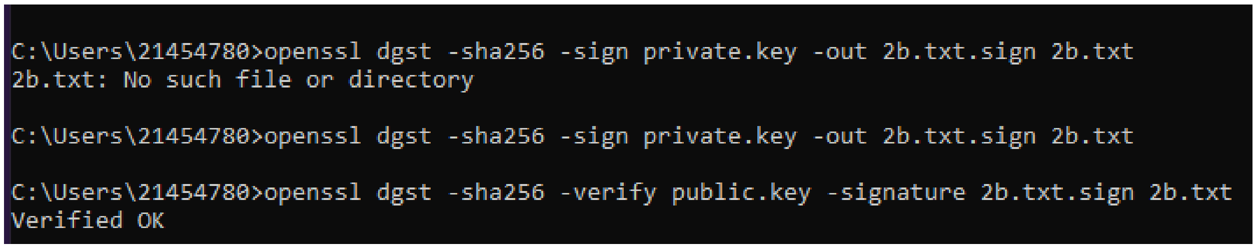
CoCalc Example:

Animated Example

Windows Example:

Animated Example

Paste a screenshot of the file being verified correctly with the public key/openssl.



```
C:\Users\21454780>openssl dgst -sha256 -sign private.key -out 2b.txt.sign 2b.txt
2b.txt: No such file or directory

C:\Users\21454780>openssl dgst -sha256 -sign private.key -out 2b.txt.sign 2b.txt

C:\Users\21454780>openssl dgst -sha256 -verify public.key -signature 2b.txt.sign 2b.txt
Verified OK
```

5. Let us modify the 2b.txt file by adding a period to the end of the file and run the verify operation again.

CoCalc:

```
echo . >> 2b.txt
openssl dgst -sha256 -verify public.key -signature 2b.txt.sign 2b.txt
```

Windows

```
echo . >> 2b.txt
openssl dgst -sha256 -verify public.key -signature 2b.txt.sign 2b.txt
```

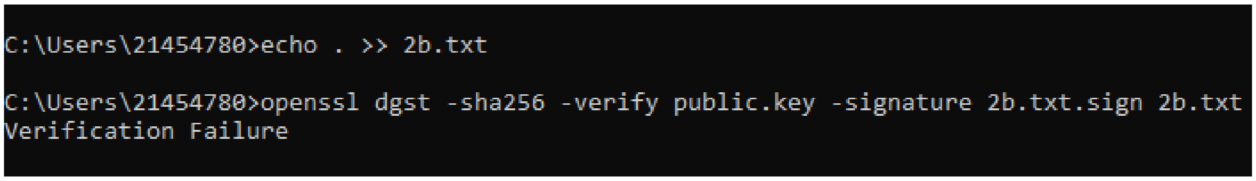
CoCalc Example:

Animated Example

Windows Example:

Animated Example

Paste a screenshot of the signature of the file not being validated after it has been modified.



```
C:\Users\21454780>echo . >> 2b.txt

C:\Users\21454780>openssl dgst -sha256 -verify public.key -signature 2b.txt.sign 2b.txt
Verification Failure
```

Lab 2 – Secure Key Storage and Encryption/Decryption with RSA and AES

In this second lab, we will look at securely storing an AES using RSA. This storage method ensures that only the person who has the private key will be able to access the AES key for encryption and decryption of the file. This is very similar to how file encryption works in real software with the exception that the RSA private key is usually also stored encrypted or with on secure hardware like a smart cart.

6. Next, we will generate AES keys using the OpenSSL rand function.

CoCalc:

Create the random AES key and store it in a file.

```
openssl rand -hex -out key.bin 32
```

Create the random IV:

```
openssl rand -hex -out iv.bin 16
```

View the files

```
cat key.bin; cat iv.bin
```

Windows:

Create the random AES key and store it in a file.

```
openssl rand -hex -out key.bin 32
```

Create the random IV:

```
openssl rand -hex -out iv.bin 16
```

View the files

```
more key.bin; more iv.bin
```

These commands do the following:

- openssl – This is the name of the command we are using. It has many cryptographic functions.
- rand – Use OpenSSL to generate random numbers
- -hex - Output the random numbers as hex encoded

- -out - Store the output in the file specified
- # - The number of random bytes to generate

CoCalc Example:

Animated Example

Windows Example:

Animated Example

7. Now let us encrypt the file using AES 256 CBC mode and store the result in an encrypted file using the following commands. We will read the key and iv from the file and store them in variables.

<u>Note: Use echo $var or %var% to make sure the key and iv are stored in variable of the same name before trying to encrypt the 2b.txt file.</u>

CoCalc:

Note: Use backticks on the upper left of your keyboard for the export command

```
export key=`cat key.bin`
export iv=`cat iv.bin`
echo $key
echo $iv
openssl enc -aes-256-cbc -in 2b.txt -out 2b.txt.enc -K $key -iv $iv
```

Windows:

```
set /p key=<key.bin
set /p iv=<iv.bin
echo %key%
echo %iv%
openssl enc -aes-256-cbc -in 2b.txt -out 2b.txt.enc -K %key% -iv %iv%
```

These commands do the following:

- openssl – This is the name of the command we are using. It has many cryptographic functions.
- enc – Use encrypt mode of OpenSSL
- -aes-256-cbc - Encrypt using AES 256 in CBC mode
- -in - The plaintext file to encrypt
- -out - The file that stores the final ciphertext
- -K <key var> - Use the key stored in the key variable
- -iv <iv var> - Use the IV stored in the iv variable

CoCalc Example:

Animated Example

Windows Example:

Animated Example

8. In this step we will encrypt the key file with our public RSA key and store the encrypted AES key in a new file. We will also delete the unencrypted key and unset the variable in memory storing the key.

Note: Make sure that the key.bin.enc and encrypted file 2b.txt.enc have been created before deleting the key.bin and 2b.txt files using the ls (Linux/CoCalc) or dir (Windows) commands.

CoCalc:

```
openssl rsautl -encrypt -inkey public.key -pubin -in key.bin -out
key.bin.enc
```

```
ls
```

```
rm key.bin
```

```
unset key
```

```
rm 2b.txt
```

Windows:

```
openssl rsautl -encrypt -inkey public.key -pubin -in key.bin -out
key.bin.enc
```

```
dir
```

```
del key.bin
```

```
set key=
```

```
del 2b.txt
```

These commands do the following:

- openssl – This is the name of the command we are using. It has many cryptographic functions.
- rsautl – Use RSA utility functions
- -encrypt - Encrypt using an RSA key (public or private)
- -inkey - The public or private key to use for encryption
- -pubin - Specify that the key being used in public
- -in - The file to encrypt (note: the data being encrypted must be under a certain size)
- -out - Store the RSA encrypted data in this file

CoCalc Example:

Animated Example

Windows Example:

Animated Example

9. Check to make sure the unencrypted key has been deleted and the unencrypted 2b.txt file has been deleted. This would be normal secure state of the computer, with nothing but the RSA encrypted AES key available until you are ready to decrypt the file.

CoCalc:

```
cat key.bin
echo $key
cat 2b.txt
```

Windows:

```
more key.bin
echo %key%
more 2b.txt
```

CoCalc Example:

Animated Example

Windows Example:

Animated Example

10. Now we will decrypt the RSA encrypted AES key and store this key in the key variable.

Note: Make sure that when you run the echo command you see the hex encoded AES key before moving on. Also note that the iv is stored unencrypted (this is usually stored with the file that was encrypted).

CoCalc:

```
export key=`openssl rsautl -decrypt -inkey private.key -in key.bin.enc`
echo $key
export iv=`cat iv.bin`
echo $iv
```

Windows:

```
openssl rsautl -decrypt -inkey private.key -in key.bin.enc -out tmp.bin
& set /p key<tmp.bin & del tmp.bin
echo %key%
set /p iv=<iv.bin
echo %iv%
```

These commands do the following:

- openssl+ other – This is the name of the command we are using. It may be embedded in other commands to store the result in a variable
- rsautl – Use RSA utility functions
- -decrypt - Encrypt using an RSA key (public or private)
- -inkey - The public or private key to use for encryption
- -in - The file to decrypt
- -out - Store the decrypted data in this file
- other commands - necessary to take the OpenSSL decrypted output and store it in a variable

CoCalc Example:

Animated Example

Windows Example:

Animated Example

11. In this final step, we use the RSA decrypted AES key to decrypt and display the file

CoCalc:

```
openssl enc -aes-256-cbc -d -K $key -iv $iv -in 2b.txt.enc
```

Windows:

```
openssl enc -aes-256-cbc -d -K %key% -iv %iv% -in 2b.txt.enc
```

CoCalc Example:

Animated Example

Windows Example:

Animated Example

What is the consequence of the key being encrypted with the RSA public key? What security purpose does this serve?

Paste a screenshot of your last three command below. The last three commands should be:

openssl rsautl -decrypt -inkey -private ….

echo %key% (or) $key

openssl enc -aes-256-cbc -d -K….

Lab 3 – Diffie-Hellman Key Exchange with Key Derivation Function

In this third lab, we will look at the process where servers and clients on the Internet securely exchange/generate a shared symmetric key for secure communication over untrusted networks. We will use a password-based key derivation function in this exercise as OpenSSL does not support the Hash-Based Key Derivation Function (HKDF) from the command-line. HKDF is the normal process used to generate keys using a Diffie-Hellman key exchange/agreement. OpenSSL does include programming libraries to support HKDF. We will simulate two different users in the lab, Bob and Alice and files belonging to each virtual user will be named accordingly. Shared files will have regular naming conventions.

12. In this first step we will generate the Diffie-Hellman public parameters. This consists of a shared generator g and prime n stored in a single file.

CoCalc:

```
openssl genpkey -genparam -algorithm DH -out dhpublic.pem
cat dhpublic.pem
openssl pkeyparam -in dhpublic.pem -text
```

Windows:

```
openssl genpkey -genparam -algorithm DH -out dhpublic.pem
more dhpublic.pem
openssl pkeyparam -in dhpublic.pem -text
```

These commands do the following:

- openssl+ other – This is the name of the command we are using. It has many cryptographic functions.
- genpkey – Generate an asymmetric key or parts of one
- -genparam - We are generating public DH parameters
- algorithm - Specify the asymmetric algorithm, Diffie-Hellman in this case
- -out - Store the resultant DH public parameters in the filename specified
- pkeyparam - Public or private key processing tools
- -in - Read the public key parameters in from a file
- -text - Display the parameters on the screen in text form

CoCalc Example:

Animated Example

Windows Example:

Animated Example

Paste the output of the openssl pkeyparam -in dhpublic.pem -text command below.

13. In the next step Bob and Alice use the public g and n (DH parameters) to generate Bob's a and Alice's b, which are essentially private keys. Normally this will be done on two separate computers on a network.

CoCalc:

Bob generates a:

```
openssl genpkey -paramfile dhpublic.pem -out bob.pem
```

Alice generates b:

```
openssl genpkey -paramfile dhpublic.pem -out alice.pem
```

Windows:

Bob generates a:

```
openssl genpkey -paramfile dhpublic.pem -out bob.pem
```

Alice generates b:

```
openssl genpkey -paramfile dhpublic.pem -out alice.pem
```

These commands do the following:

- openssl+ other – This is the name of the command we are using. It has many cryptographic functions.
- genpkey – Generate an asymmetric key or parts of one (private DH key in this case)
- -paramfile - The DH public parameters needed to create a private key
- -out - Output the private key to the filenames specified

CoCalc Example:

Animated Example

Windows Example:

Animated Example

14. In the next step Bob and Alice use the public g and n (DH parameters) to generate Bob's public key (g^a mod 17) and Alice's public key (g^b mod 17). These numbers will normally be exchanged over a network.

CoCalc:

Bob generates g^a mod n and gives it to Alice:

```
openssl pkey -in bob.pem -pubout -out bobpublic.pem
```

Alice generates g^b mod n and gives it Bob:

```
openssl pkey -in alice.pem -pubout -out alicepublic.pem
```

Windows:

Bob generate a:

```
openssl pkey -in bob.pem -pubout -out bobpublic.pem
```

Alice generate b:

```
openssl pkey -in alice.pem -pubout -out alicepublic.pem
```

These commands do the following:

- openssl+ other – This is the name of the command we are using. It has many cryptographic functions.
- pkey – Generate an asymmetric key or parts of one (public DH key in this case)
- -in - Private key file
- -pubout - Output only the public key
- -out - Output the public key to the filename specified

CoCalc Example:

Animated Example

Windows Example:

Animated Example

15. Now Alice and Bob can calculate/derive g^{ab} mod n and g^{ba} mod n separately and come up with the same shared secret.

CoCalc:

Bob generates g^{ab} mod n to get shared secret:

```
openssl pkeyutl -derive -inkey bob.pem -peerkey alicepublic.pem |
openssl base64 > bobsharedsecret.b64
```

Alice generates g^{ba} mod n to get shared secret:

```
openssl pkeyutl -derive -inkey alice.pem -peerkey bobpublic.pem |
openssl base64 > alicesharedsecret.b64
```

Check to see if shared secrets are the same:

```
cat bobsharedsecret.b64; echo .; cat alicesharedsecret.b64;
```

Windows:

Bob generates g^{ab} mod n to get shared secret:

```
openssl pkeyutl -derive -inkey bob.pem -peerkey alicepublic.pem |
openssl base64 > bobsharedsecret.b64
```

Alice generates g^{ba} mod n to get shared secret:

```
openssl pkeyutl -derive -inkey alice.pem -peerkey bobpublic.pem |
openssl base64 > alicesharedsecret.b64
```

Check to see if shared secrets are the same:

```
more bobsharedsecret.b64 & echo . & more alicesharedsecret.b64
```

These commands do the following:

- openssl+ other – This is the name of the command we are using. It has many cryptographic functions.
- pkeyutl – Public key utility functionality
- -derive - Use public and private keys to derive a shared secret
- -inkey - The file with your private key
- -outkey - The file with your peer’s public key
- base64 - Output the result in Base64

CoCalc Example:

Animated Example

Windows Example:

Animated Example

Paste the last cat or more command below to show that the shared secrets are the same:

16. Finally, we will use these shared secrets in a key derivation function to generate shared AES 256 keys. Notice that Bob and Alice arrive at the same AES key, yet anyone sniffing the key exchange would not be able to calculate or derive the AES key.

Note: This is only used as an example as you should not use PBKDF2 to generate a symmetric key from DH shared secrets but rather use HKDF. HKDF does not work from the command line using OpenSSL so this is only being done this way as an example.

CoCalc:

Bob generates symmetric key using KDF and shared secret as a seed:

```
openssl enc -aes-256-cbc -nosalt -kfile bobsharedsecret.b64 -P
```

Alice generates symmetric key using KDF and shared secret as a seed:

```
openssl enc -aes-256-cbc -nosalt -kfile alicesharedsecret.b64 -P
```

Windows:

Bob generates symmetric key using KDF and shared secret as a seed:

```
openssl enc -aes-256-cbc -nosalt -kfile bobsharedsecret.b64 -P
```

Alice generates symmetric key using KDF and shared secret as a seed:

```
openssl enc -aes-256-cbc -nosalt -kfile alicesharedsecret.b64 -P
```

These commands do the following:

- openssl+ other – This is the name of the command we are using. It has many cryptographic functions.
- enc – Use OpenSSL encryption mode
- -aes-256-cbc - Use AES 256 in CBC mode
- -nosalt - Do not add a random salt in creating the key (We are using this simulate HKDF which does not use a random salt. Random salts are only used for password-based KDFs)
- -kfile - Use the shared secret we derived from Diffie-Hellman as the seed to generate the key
- -P - Do not encrypt anything, just show the key and iv that would be used

CoCalc Example:

Animated Example

Windows Example:

Animated Example

Paste the output of the last two Alice and Bob commands generating the AES 256 CBC key and IV and make sure both the key and iv are the same between Bob and Alice.