Overview:

The objective of this lab was to explore and understand the basic principles of public-key cryptography by using OpenSSL to generate key pairs, encrypt/decrypt messages, and sign/verify messages with RSA keys.

Lab Environment

I performed this lab on a Kali Linux virtual machine where OpenSSL was already installed. OpenSSL libraries and commands are integral for working with public-key cryptography in this lab. If you want to use OpenSSL in a programming environment, additional installation of libraries, header files, and manuals would be required.

I also installed the GHex hex editor on the VM, which allowed me to view and modify binary files for tasks involving encryption and decryption of messages.

Please include the screenshots of the results for each of the following questions.

Task 1: Generating RSA Public/Private key pair (20 points)

The first task is to generate an RSA key pair. This is done in two phases in OpenSSL; first you generate your private key, as follows:

% openssl genrsa -aes128 -out private_key.pem 1024 A
couple things to note here about this input:

- genrsa: As you've come to see by now, OpenSSL's command line tool takes a command as a first argument (dgst, genrsa, and later we will see rsautl).

- -out private_key.pem: Ultimately, the goal of the above command is to generate a private key. This argument specifies the name of the file to save it to. I've given this a .pem file extension to denote the fact that OpenSSL defaults to the PEM file format when working with public and private keys. (The name comes from Privacy Enhanced eMail, but the only thing that stuck from that system was the file format, so nobody ever calls it anything but PEM.)

- -aes128: This option informs OpenSSL not to store your private key in plaintext, but rather to first encrypt it with 128-bit AES before writing it to disk. A natural question to ask is: but what is the key, and won't I need to store that key, and a key to encrypt that key, and a key to encrypt that key, and...

After you run the command, you'll notice that it asks you for a pass phrase. OpenSSL uses "passwordbased encryption," using your password as input to determine an AES key. What this means is that every time you use your private key, you must be able to provide your

password so that OpenSSL can decrypt the part of your private_key.pem that contains the private key. Of course, as with the dgst command, you get many options here for encrypting (if you Google for examples, you will see that des3 is a common option, despite 3DES's shortcomings).

• 1024: Finally, we provide our desired key size.

Now that we have our private key, we can generate the corresponding public key:

% openssl rsa -in private_key.pem -out public_key.pem -pubout

Note that this is using the rsa command as opposed to the genrsa command we used to generate our private key in the first place. Also, since we encrypted the private key, we should expect this command to ask us for our pass phrase. Most of the arguments are pretty straightforward. -in denotes the input file: our private key, while -out denotes the output file: our public key. The -pubout option makes explicit that we will be generating a public key as our output.

Analysis:

I generated an RSA key pair by first creating a private key using OpenSSL. The following command was used to generate the private key:

openssl genrsa -aes128 -out private_key.pem 1024

Here's what each part of the command did:

- *genrsa*: OpenSSL's command to generate RSA keys.
- *-aes128*: Encrypts the private key using 128-bit AES.
- *-out private_key.pem*: Specifies the filename to save the private key.
- *1024*: Specifies the key length (1024 bits in this case).

I was prompted for a passphrase to encrypt the private key. After successfully generating the private key, I created the corresponding public key with this command:

openssl rsa -in private_key.pem -out public_key.pem -pubout

After running this command and entering my passphrase, the public key was successfully created.

Public Key Encryption/Decryption

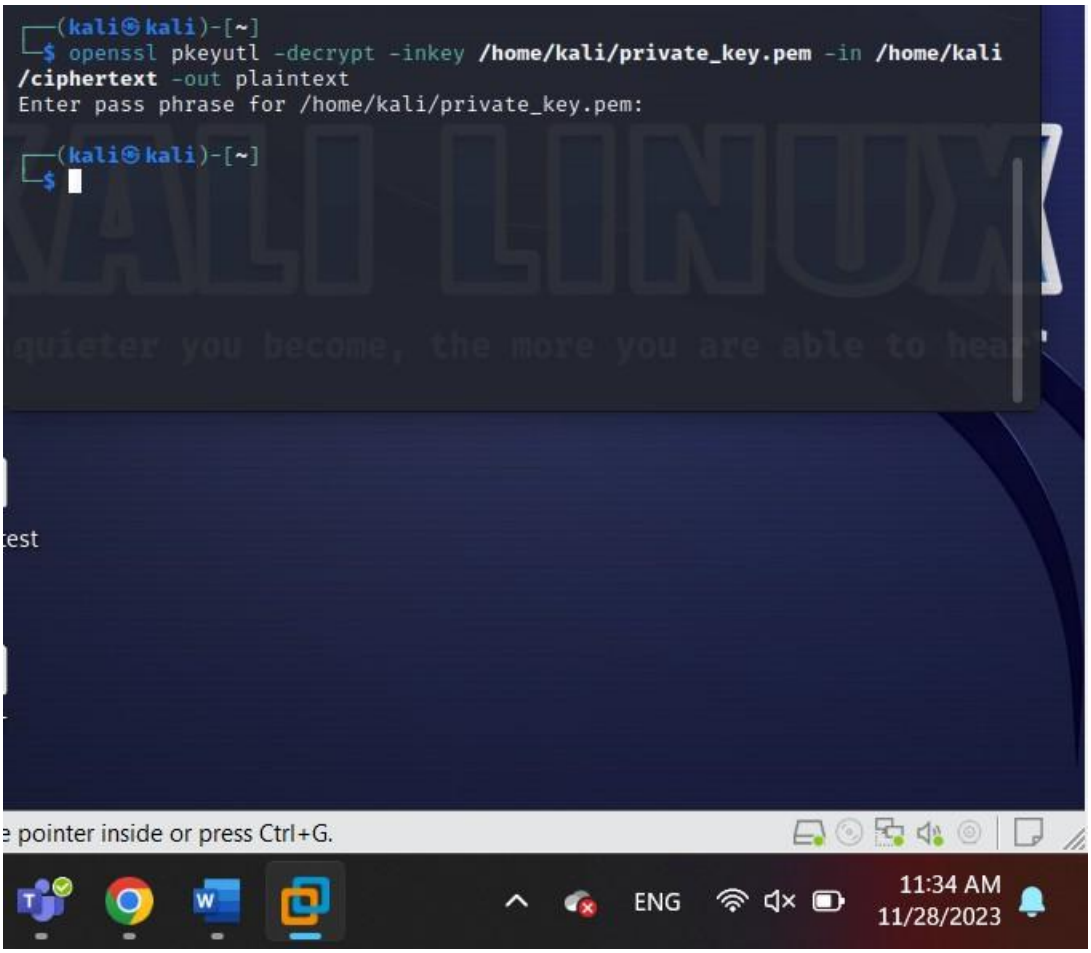
Next, I explored how public-key encryption and decryption work in OpenSSL. To encrypt a file, I used the following command:

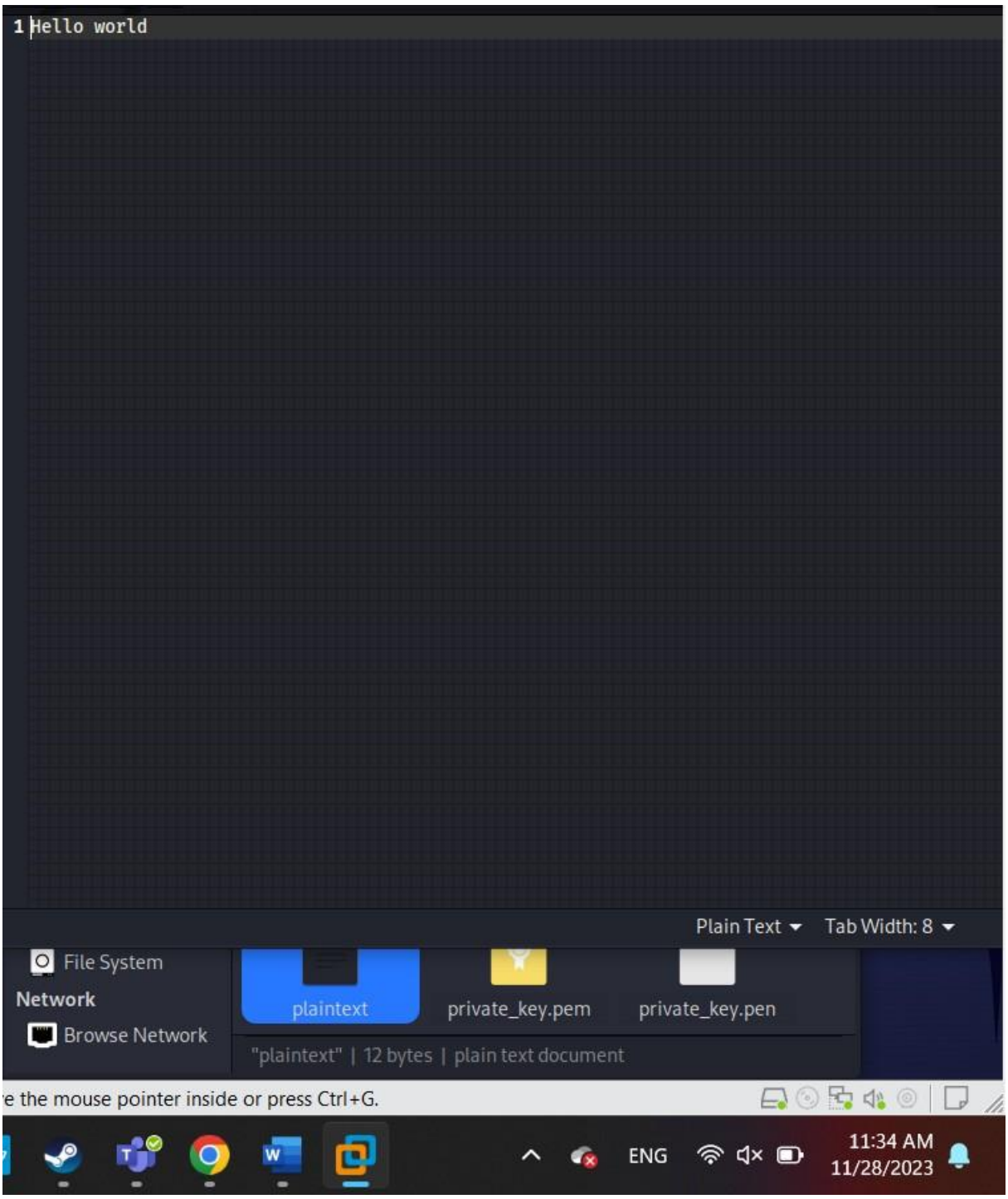
% *openssl pkeyutl -encrypt -inkey public_key.pem -pubin -in plaintext -out ciphertext*

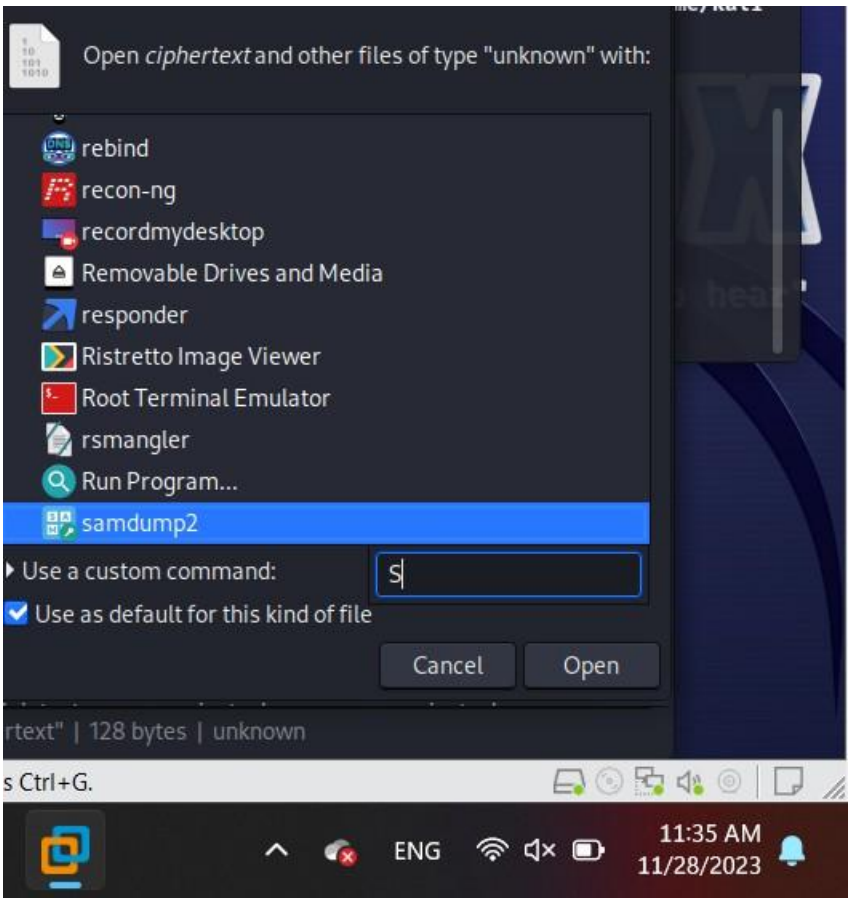
Decryption works similarly:

% *openssl pkeyutl -decrypt -inkey private_key.pem -in ciphertext -out plaintext*

After entering my passphrase for the private key, the original plaintext file was successfully restored.


(kali㉿kali)-[~]
$ openssl pkeyutl -decrypt -inkey /home/kali/private_key.pem -in /home/kali
/ciphertext -out plaintext
Enter pass phrase for /home/kali/private_key.pem:
(kali㉿kali)-[~]
$
KALI LINUX
quieter you become, the more you are able to hear
test
e pointer inside or press Ctrl+G.
ENG
11:34 AM
11/28/2023


1 Hello world
Plain Text
Tab Width: 8
File System
Network
Browse Network
plaintext
private_key.pem
private_key.pen
"plaintext" | 12 bytes | plain text document
e the mouse pointer inside or press Ctrl+G.
ENG
11:34 AM
11/28/2023



Public key signatures/verification

I finally performed signatures and signature verification. Like with encrypt/decrypt, rsautl provides a way to sign and verify, but also like encrypt/decrypt, it only operates over inputs no larger than the key.

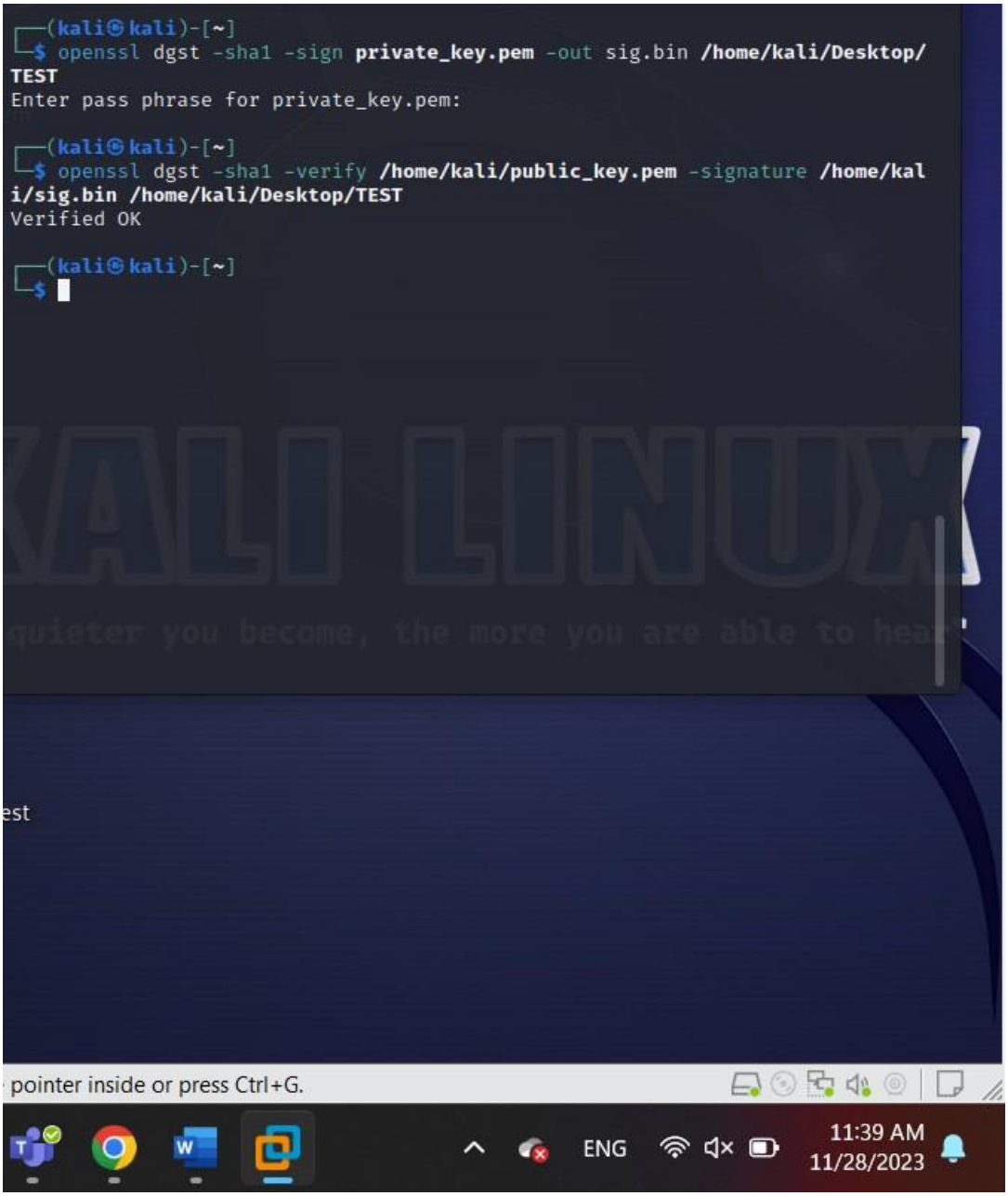
I signed the plaintext file with the following command:

% *openssl dgst -sha1 -sign private_key.pem -out sig.bin plaintext*

To verify the signature, I used this command:

% *openssl dgst -sha1 -verify public_key.pem -signature sig.bin plaintext*

The signature was verified successfully, confirming that the file had not been tampered with and was indeed signed by the holder of the corresponding private key.


(kali㉿kali)-[~]
$ openssl dgst -sha1 -sign private_key.pem -out sig.bin /home/kali/Desktop/TEST
Enter pass phrase for private_key.pem:
(kali㉿kali)-[~]
$ openssl dgst -sha1 -verify /home/kali/public_key.pem -signature /home/kali/sig.bin /home/kali/Desktop/TEST
Verified OK
(kali㉿kali)-[~]
$
KALI LINUX
quieter you become, the more you are able to hea
est
pointer inside or press Ctrl+G.
ENG
11:39 AM
11/28/2023