Lab2 – ECE 1155

Rayan Hassan – 4511021

There are different encryption methods used nowadays that are very efficient. RSA is an example of that. In RSA, messages are encrypted using a public key (e,n) based on two large prime numbers and decrypted using a private key (d,n) known only by those who receive the message. For encryption, $message^e mod\ n$ is used, and for decryption $ciphertext^d mod\ n$ is computed to retrieve the message. Digital signatures use a similar concept, where an entity signs a message using their private key $signature = \ message^e mod\ n$, and the receiver verifies the signature using the sender's public key. All in all, these methods are secure and widely used in today's world.

Task 1

The following command window shows the values I got for n (n= p x q), $\varphi(n)$, and the private key d.

For question c), to validate that the public key e is relatively prime to $\varphi(n)$, I calculated the gcd(e,$\varphi(n)$) which is also shown below.

```
/bin/bash 105x24
[03/04/22]seed@VM:~$ cd Desktop
[03/04/22]seed@VM:~/Desktop$ gcc Q1.c -o Q1_sample -lcrypto
[03/04/22]seed@VM:~/Desktop$ ./Q1_sample
n =  E103ABD94892E3E74AFD724BF28E78366D9676BCCC70118BD0AA1968DBB143D1
Totient(n) =  E103ABD94892E3E74AFD724BF28E78348D52298BD687C44DEB3A81065A7981A4
Private key 'd' =  C5148B1AD7C6D85847C52EABB879F26CCCEE8EB70D2282DC6050309C18F85
D85
gcd(e, totient(n)) =  01
[03/04/22]seed@VM:~/Desktop$
```

My C code is shown below. I changed 'a', 'b', … to match the values of the task 'p', 'q' …

```c
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
void printBN(char *msg, BIGNUM * a)
{
/* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
char * number_str = BN_bn2hex(a);
printf("%s %s\n", msg, number_str);
OPENSSL_free(number_str);
}
int main ()
{
BN_CTX *ctx = BN_CTX_new();

BIGNUM *p = BN_new();
BIGNUM *q = BN_new();
BIGNUM *n = BN_new();
BIGNUM *res = BN_new();
BIGNUM *res1 = BN_new();
BIGNUM *res2 = BN_new();
BIGNUM *res3 = BN_new();
BIGNUM *e = BN_new();
BIGNUM *d = BN_new();

// Initialize p, q, e
BN_hex2bn(&p,  "F7E75FDC469067FFDC4E847C51F452DF");
BN_hex2bn(&q,  "E85CED54AF57E53E092113E62F436F4F");
BN_dec2bn(&e,  "65537");
BN_rand(n, NBITS, 0, 0);

// res = a*b
BN_mul(res, p, q, ctx);
printBN("n = ", res);

// totient(n)=(p-1)(q-1)
BN_sub(res1, p, BN_value_one( ));
BN_sub(res2, q, BN_value_one( ));
BN_mul(res, res1, res2, ctx);
printBN("Totient(n) = ", res);


// Find d such that (ed) mod totient(n) = 1
BN_sub(res1, p, BN_value_one( ));
BN_sub(res2, q, BN_value_one( ));
BN_mul(res, res1, res2, ctx);
BN_mod_inverse(d, e, res, ctx);
printBN("Private key 'd' = ", d);

// gcd(e, totient(n))
BN_sub(res1, p, BN_value_one( ));
BN_sub(res2, q, BN_value_one( ));
BN_mul(res, res1, res2, ctx);
BN_gcd(res3, e, res, ctx);
printBN("gcd(e, totient(n)) = ", res3);


return 0;
}
```

Task 2

The command window below shows the conversion from ASCII string to a hex string

```
[03/04/22]seed@VM:~/Desktop$ python -c 'print("A top secret!".encode("hex"))'
4120746f702073656372657421
```

Here is the code that shows how to encrypt the plaintext. The formula is $C = M^e mod\ n$ where C is the ciphertext, M is a plaintext, e is the public key and n is p x q.

```c
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
void printBN(char *msg, BIGNUM * a)
{
/* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
char * number_str = BN_bn2hex(a);
printf("%s %s\n", msg, number_str);
OPENSSL_free(number_str);
}
int main ()
{
BN_CTX *ctx = BN_CTX_new();

BIGNUM *p = BN_new();
BIGNUM *q = BN_new();
BIGNUM *n = BN_new();
BIGNUM *res = BN_new();
BIGNUM *res1 = BN_new();
BIGNUM *res2 = BN_new();
BIGNUM *res3 = BN_new();
BIGNUM *e = BN_new();
BIGNUM *d = BN_new();
BIGNUM *Plaintext = BN_new();

// Initialize p, q, e
BN_hex2bn(&p,  "F7E75FDC469067FFDC4E847C51F452DF");
BN_hex2bn(&q,  "E85CED54AF57E53E092113E62F436F4F");
BN_hex2bn(&Plaintext,   "4120746f702073656372657421");
BN_dec2bn(&e,  "65537");
BN_rand(n, NBITS, 0, 0);


// Calculate C=M^e mod n
BN_mul(res1, p, q, ctx);
BN_mod_exp(res, Plaintext, e, res1, ctx);
printBN("Ciphertext = ", res);
return 0;
}
```

The ciphertext obtained is shown in the following command window:

```
[03/04/22]seed@VM:~/Desktop$ gcc Q1.c -o Q1_sample -lcrypto
[03/04/22]seed@VM:~/Desktop$ ./Q1_sample
Ciphertext =  951F9D0B1D0BFEA47FB95B12F8EBDD5773451E3ACC1EB5701DDA639376606918
[03/04/22]seed@VM:~/Desktop$
```

Task 3

```c
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
void printBN(char *msg, BIGNUM * a)
{
/* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
char * number_str = BN_bn2hex(a);
printf("%s %s\n", msg, number_str);
OPENSSL_free(number_str);
}
int main ()
{
BN_CTX *ctx = BN_CTX_new();

BIGNUM *p = BN_new();
BIGNUM *q = BN_new();
BIGNUM *n = BN_new();
BIGNUM *res = BN_new();
BIGNUM *res1 = BN_new();
BIGNUM *res2 = BN_new();
BIGNUM *res3 = BN_new();
BIGNUM *res4 = BN_new();
BIGNUM *res5 = BN_new();
BIGNUM *res6 = BN_new();
BIGNUM *e = BN_new();
BIGNUM *d = BN_new();
BIGNUM *Plaintext = BN_new();

// Initialize p, q, e
BN_hex2bn(&p,  "F7E75FDC469067FFDC4E847C51F452DF");
BN_hex2bn(&q,  "E85CED54AF57E53E092113E62F436F4F");
BN_hex2bn(&Plaintext,   "4120746f702073656372657421");
BN_dec2bn(&e,  "65537");
BN_rand(n, NBITS, 0, 0);


// Calculate M=C^d mod n
BN_mul(res1, p, q, ctx); // this is 'n'
BN_mod_exp(res2, Plaintext, e, res1, ctx); //calculate ciphertext
BN_sub(res3, p, BN_value_one( ));
BN_sub(res4, q, BN_value_one( ));
BN_mul(res5, res3, res4, ctx);
BN_mod_inverse(d, e, res5, ctx); // to find private key 'd'
BN_mod_exp(res6, res2, d, res1, ctx);
printBN("Recovered plaintext = ", res6);
return 0;
}
```

To obtain the original message, we should decrypt it following this formula: $M = C^d mod\ n$ where d is the private key. This is shown in the C code above.

Below is the command window that shows the recovered message in hexadecimal and in ASCII string. The recovered message is in fact: “A top secret!”.

```
[03/04/22]seed@VM:~/Desktop$ gcc Q1.c -o Q1_sample -lcrypto
[03/04/22]seed@VM:~/Desktop$ ./Q1_sample
Recovered plaintext =  4120746F702073656372657421
```

```
[03/04/22]seed@VM:~/Desktop$ python -c 'print("4120746F702073656372657421".decode("hex"))'
A top secret!
[03/04/22]seed@VM:~/Desktop$
```

Task 4

a) To sign the message “I owe you $2000”, we need to convert it to a hex string:

```
[03/04/22]seed@VM:~/Desktop$ python -c 'print("I owe you $2000".encode("hex"))'
49206f776520796f75202432303030
```

The following code shows how the message is signed. The signature is done using the private key: $signature = message^d mod\ n$. The code below shows how I did that

```c
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
void printBN(char *msg, BIGNUM * a)
{
/* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
char * number_str = BN_bn2hex(a);
printf("%s %s\n", msg, number_str);
OPENSSL_free(number_str);
}
int main ()
{
BN_CTX *ctx = BN_CTX_new();

BIGNUM *p = BN_new();
BIGNUM *q = BN_new();
BIGNUM *n = BN_new();
BIGNUM *res = BN_new();
BIGNUM *res1 = BN_new();
BIGNUM *res2 = BN_new();
BIGNUM *res3 = BN_new();
BIGNUM *res4 = BN_new();
BIGNUM *res5 = BN_new();
BIGNUM *res6 = BN_new();
BIGNUM *e = BN_new();
BIGNUM *d = BN_new();
BIGNUM *Plaintext = BN_new();

// Initialize p, q, e
BN_hex2bn(&p,  "F7E75FDC469067FFDC4E847C51F452DF");
BN_hex2bn(&q,  "E85CED54AF57E53E092113E62F436F4F");
BN_hex2bn(&Plaintext,   "49206f776520796f75202432303030");
BN_dec2bn(&e,  "65537");
BN_rand(n, NBITS, 0, 0);


// Calculate signature = plaintext^d mod n
BN_mul(res1, p, q, ctx); // this is 'n'
BN_sub(res3, p, BN_value_one( ));
BN_sub(res4, q, BN_value_one( ));
BN_mul(res5, res3, res4, ctx);
BN_mod_inverse(d, e, res5, ctx); // to find private key 'd'
BN_mod_exp(res6, Plaintext, d, res1, ctx);
printBN("Signed message = ", res6);
return 0;
}
```

The signed message is shown as hex string and ASCII string below:

```
[03/04/22]seed@VM:~/Desktop$ gcc Q1.c -o Q1_sample -lcrypto
[03/04/22]seed@VM:~/Desktop$ ./Q1_sample
Signed message =  D9A54876C4449F6373924691599AA135A226332651C3F0493750A1967748BDDB
```

```
[03/04/22]seed@VM:~/Desktop$ python -c 'print("D9A54876C4449F6373924691599AA135A226332651C3F0493750A19677
48BDDB".decode("hex"))'
0Hv�D�cs�F�Y��5�&3&Q��I7P��wH�
```

b) After changing the message to “I owe you $3000”, the corresponding hex string (which I use to initialize the plaintext) and final sigature (hex string and ASCII string) are shown below.

```
[03/04/22]seed@VM:~/Desktop$ python -c 'print("I owe you $3000".encode("hex"))'
49206f776520796f75202433303030
[03/04/22]seed@VM:~/Desktop$ gcc Q1.c -o Q1_sample -lcrypto
[03/04/22]seed@VM:~/Desktop$ ./Q1_sample
Signed message =  7BBCA97A436293FFA22403201A46E207CD23D9943EB2E0C9F57B08F404529830
```

```
[03/04/22]seed@VM:~/Desktop$ python -c 'print("7BBCA97A436293FFA22403201A46E207CD23D9943EB2E0C9F57B08F404
529830".decode("hex"))'
{��zCb���$ F��#>�����R�0
[03/04/22]seed@VM:~/Desktop$
```

Task 5

1) To verify whether the signature is indeed Alice’s: $message = signature^{e} mod\ n$, where e is Alice’s public key. The code below shows how I implemented that operation:

```c
/* bn_sample.c */
#include <stdio.h>
#include <openssl/bn.h>
#define NBITS 256
void printBN(char *msg, BIGNUM * a)
{
/* Use BN_bn2hex(a) for hex string
* Use BN_bn2dec(a) for decimal string */
char * number_str = BN_bn2hex(a);
printf("%s %s\n", msg, number_str);
OPENSSL_free(number_str);
}
int main ()
{
BN_CTX *ctx = BN_CTX_new();

BIGNUM *p = BN_new();
BIGNUM *q = BN_new();
BIGNUM *n = BN_new();
BIGNUM *res = BN_new();
BIGNUM *res1 = BN_new();
BIGNUM *res2 = BN_new();
BIGNUM *res3 = BN_new();
BIGNUM *res4 = BN_new();
BIGNUM *res5 = BN_new();
BIGNUM *res6 = BN_new();
BIGNUM *e = BN_new();
BIGNUM *d = BN_new();
BIGNUM *Plaintext = BN_new();
BIGNUM *Signature = BN_new();

// Initialize p, q, e
BN_hex2bn(&p,  "F7E75FDC469067FFDC4E847C51F452DF");
BN_hex2bn(&q,  "E85CED54AF57E53E092113E62F436F4F");
BN_hex2bn(&Signature,   "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
BN_dec2bn(&e,  "65537");
BN_hex2bn(&n,   "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");


// Calculate message = signature^e mod n

BN_mod_exp(res, Signature, e, n, ctx);
printBN("Verified message = ", res);
return 0;
}
```

The following command window verifies Alice’s signature as the message is in fact “Launch a missile”

```
[03/04/22]seed@VM:~/Desktop$ gcc Q1.c -o Q1_sample -lcrypto
[03/04/22]seed@VM:~/Desktop$ ./Q1_sample
Verified message =  4C61756E63682061206D697373696C652E
[03/04/22]seed@VM:~/Desktop$ python -c 'print("4C61756E63682061206D697373696C652E".decode("hex"))'
Launch a missile.
[03/04/22]seed@VM:~/Desktop$
```

I changed the last bit of the signature from 2F to 3F, as shown below

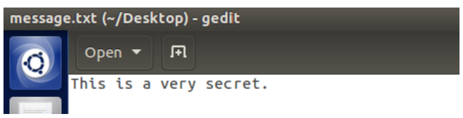
```
// Initialize p, q, e
BN_hex2bn(&p,  "F7E75FDC469067FFDC4E847C51F452DF");
BN_hex2bn(&q,  "E85CED54AF57E53E092113E62F436F4F");
BN_hex2bn(&Signature,   "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");
BN_dec2bn(&e,  "65537");
BN_hex2bn(&n,   "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
```

The corresponding output is the following, showing that the verification process completely changes since the signature is not Alice's. Therefore, we can't retrieve the original message. As we can see, the final output is very different:

```
[03/04/22]seed@VM:~/Desktop$ gcc Q1.c -o Q1_sample -lcrypto
[03/04/22]seed@VM:~/Desktop$ ./Q1_sample
Verified message =  91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
[03/04/22]seed@VM:~/Desktop$ python -c 'print("91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294"
.decode("hex"))'
��,�D�c���rm=f�:N�����
[03/04/22]seed@VM:~/Desktop$
```

Task 6

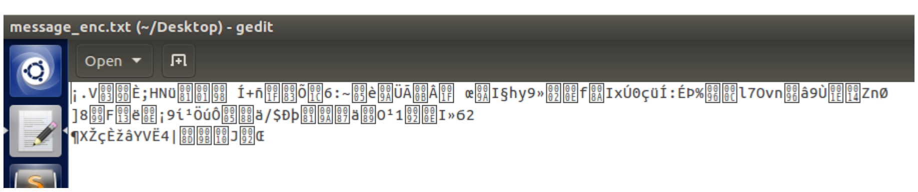
1) The message.txt file is the following:



2) The command window shows how I created the public and private key (private.pem, public.pem)

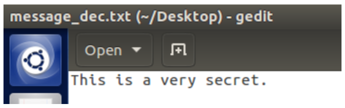
```
[03/04/22]seed@VM:~$ cd Desktop
[03/04/22]seed@VM:~/Desktop$ openssl genrsa -out private.pem 1024
Generating RSA private key, 1024 bit long modulus
....++++++
.++++++
e is 65537 (0x10001)
[03/04/22]seed@VM:~/Desktop$ more private.pem
-----BEGIN RSA PRIVATE KEY-----
MIICXAIBAAKBgQDH4rm5vZZ0mHJrO490ptwr/NCNZMT8T5Olubg5aNLe20xiR/Gr
B3ZIZWH97f0S2Ca/sG5p/u1L8rnl1V4OrUnjEuLWek6zIFMqXtz8CckxEbVArmkc
h4JWRXXcCibPBt4J70WIaHPGeJHrucSsgIpSJRKVqi8/C2eVJf5NsZKgSQIDAQAB
AoGALR4/P+7MbEsQ4KWUDwevAPrS9Gd/k1g/ApR2QfcoSgeEKVcF9M39ZAAGa3Px
wPQrYu9nLOPPGmqSjY9o6LUyzPmOM2IiFThwdSiPKNpKLCXAz+ojTSMicTfzd3Y3
BfZKBdDlViZUkZ4k7wxHezUPQnIjVQfBcATGosu4BbmfdFECQQDmO0v+9DpO6Ho6
arTWLumJNrGYNZrDE0BgpGXVYqIi9Tb1QujP/+pmoqv9gprj6whFm7AnAbH4uf7V
SnDE7RoFAkEA3kHxB0KaADvQgNon41++peoVaalr9HJVNTC4pbplCokbDisif/zM
gAXClBU4zKMI/N7LtVsRghRWVKsvbP+MdQJAEKM4WrRLgmhKbO5JIzFArynEq7sz
8+TwQgufUExQazZNQmwTZvSXe9NFmoxVpSLW9jDKCgeVVOQcBk3a2GBKAQJBAIeR
g1qhVlxEH+coqmH3mcuMICmGLRI61xyNrmNwcYdeoAEhcHbPFVIVsV0wIKg53haY
M3tYpDJLA2m7zT7YFa0CQCLmdkpxWk23esybJpYlH4tfpp6iWTbsohkktiv/hgKG
6PqNjji/QCLJ7BoRL3Q4Fj7pUWF3f6KY8Tu5AdNBp+E=
-----END RSA PRIVATE KEY-----
[03/04/22]seed@VM:~/Desktop$ openssl rsa -in private.pem -pubout > public.pem
writing RSA key
```

3) The encrypted message is the following



4) The decrypted ciphertext (original plaintext) is shown below as well as the corresponding command

```
[03/04/22]seed@VM:~$ cd Desktop
[03/04/22]seed@VM:~/Desktop$ openssl rsautl -decrypt -inkey private.pem -in message_enc.txt -out message_dec.txt
```



Task 7

Speed of RSA:

```
[03/04/22]seed@VM:~/Desktop$ openssl speed rsa
Doing 512 bit private rsa's for 10s: 64759 512 bit private RSA's in 9.85s
```

Therefore 64759 blocks of data are encrypted with 512-bits RSA in 9.85 seconds. So the time to sign with 512 bit key of RSA is 9.85/64759 = 0.0001521 seconds

Speed of AES:

```
[03/04/22]seed@VM:~/Desktop$ openssl speed aes
Doing aes-128 cbc for 3s on 16 size blocks: 15181869 aes-128 cbc's in 2.90s
```

So 15181869 blocks of data each of 1024 bytes are encrypted in 2.9 seconds.

So we can tell that AES is much faster than RSA.

In conclusion, we can see that RSA is very secure. For instance, an error in the signature results in a complete failure of the verification process, so only a signature coming from a legitimate user can be verified. In addition, if a signed message slightly changes, the corresponding signature will change drastically. Finally, although RSA is much slower than AES, it is more secure since it is much more computationally intensive. However, both have their pros and cons and their use depends on what we would like to achieve.