Lab 3 – RSA Public Key Encryption and Signature(Seed Labs)

Taylor Adams, CS 542 – Cryptography, tkadams1@crimson.ua.edu

***Abstract*—The goal of the lab is to allow the user to gain a greater understanding RSA Public key encryption and the bignums library. This lab demonstrates the use of secure and non-secure was to generate randomness. This lab also has the user demonstrate signing a certificate and manually verifying a X.509 certificate from a certificate authority. This lab gives the user a hands on understanding of the importance of RSA encryption in web security.**

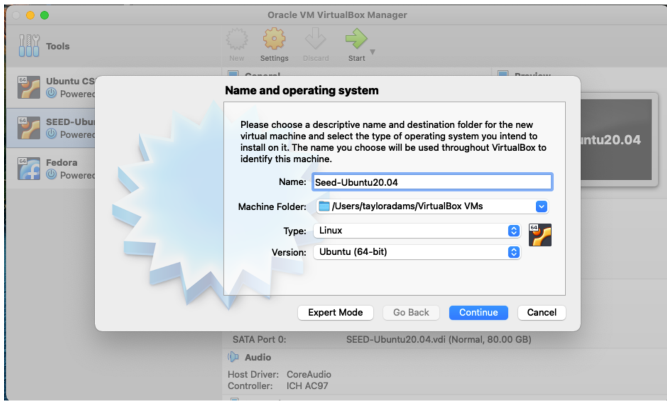
I. Lab Definition

THE goal of this lab is to give students the opportunity to learn about how RSA works. It also gives the student the tools necessary to understand how to utilize the bignums library. This lab shows several popular libraries used in pseudo random number generation and ask the student to perform task with theses libraries. In task 1, the student is asked to get the private key given p, q and e. Tasks 2 and 3 has the student encrypt and decrypt a message using RSA encryption and the Bignums library. Task 4 has the student sign a message, to ensure the integrity of the message through RSA encryption. The student is then asked to modify the signed message as a proof that RSA will preserve the integrity of the original message. Task 5 provides the student with a signature and ask that the student verify the authentication of a message signed with the private key of the provided signature's owner. Task 6 goes into a "real-world" scenario and has the student verify a certification authorities signature for a website certificate on a website of their choosing.

II. Step-by-Step Instructions - Lab Task

A. *Task A – Setting up the Virtual Machine*

In order to begin this lab, I chose to utilize VirtualBox along with the pre-built VM provided at https://seedsecuritylabs.org/labsetup.html [10]. I had previously configured this VM from lab one, below are the steps I took to set up this VM This VM utilizes Ubuntu 20.04. After downloading the SEED-Ubuntu20.04.vdi file, I proceeded to create a new virtual machine to utilize the image with. I followed the setup instructions provided in the above URL to complete the setup of the lab. I have frequently utilized VirtualBox in the past, so this setup task was no problem at all.



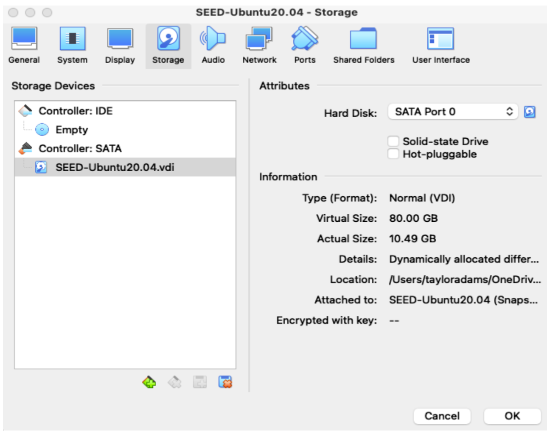
Name and operating system

Please choose a descriptive name and destination folder for the new virtual machine and select the type of operating system you intend to install on it. The name you choose will be used throughout VirtualBox to identify this machine.

Name: Seed-Ubuntu20.04
Machine Folder: /Users/tayloradams/VirtualBox VMs
Type: Linux
Version: Ubuntu (64-bit)

SATA Port 0: SEED-Ubuntu20.04.vdi (Normal, 80.00 GB)
Audio
Host Driver: CoreAudio
Controller: ICH AC97

Fig. 1. Creating the Virtual Machine

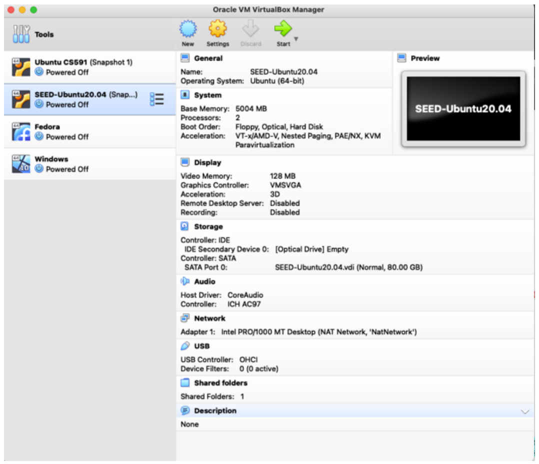


Storage Devices
Controller: IDE
Empty
Controller: SATA
SEED-Ubuntu20.04.vdi

Attributes
Hard Disk: SATA Port 0
Solid-state Drive
Hot-pluggable

Information
Type (Format): Normal (VDI)
Virtual Size: 80.00 GB
Actual Size: 10.49 GB
Details: Dynamically allocated differ...
Location: /Users/tayloradams/OneDriv...
Attached to: SEED-Ubuntu20.04 (Snaps...
Encrypted with key: --

Fig. 1a. Pre-Built VDI file placed in the virtual disk



Ubuntu CS591 (Snapshot 1) Powered Off
SEED-Ubuntu20.04 (Snap...) Powered Off
Fedora Powered Off
Windows Powered Off

General
Name: SEED-Ubuntu20.04
Operating System: Ubuntu (64-bit)

System
Base Memory: 5004 MB
Processors: 2
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, PAE/NX, KVM Paravirtualization

Display
Video Memory: 128 MB
Graphics Controller: VMSVGA
Acceleration: 3D
Remote Desktop Server: Disabled
Recording: Disabled

Storage
Controller: IDE
IDE Secondary Device 0: [Optical Drive] Empty
Controller: SATA
SATA Port 0: SEED-Ubuntu20.04.vdi (Normal, 80.00 GB)

Audio
Host Driver: CoreAudio
Controller: ICH AC97

Network
Adapter 1: Intel PRO/1000 MT Desktop (NAT Network, 'NatNetwork')

USB
USB Controller: OHCI
Device Filters: 0 (0 active)

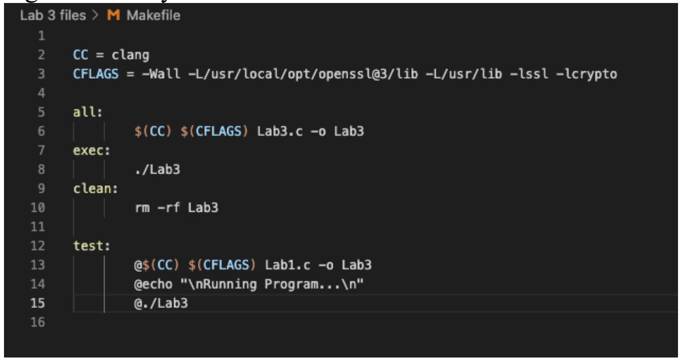
Shared folders
Shared Folders: 1

Description
None

Fig. 1b. SEED Virtual Machine Complete Settings

Manuscript received April 11, 2022.

I also installed the bignum library on my host machine (through openssl). This took around an hour to get set up. I did learn a lot from installing the bignum library on my host machine as opposed to having it come preinstalled on a vm. After installing the bignums library, I created a makefile to run on my host machine which takes in the necessary flags for the bignums library.



```
Lab 3 files > M Makefile

CC = clang
CFLAGS = -Wall -L/usr/local/opt/openssl@3/lib -L/usr/lib -lssl -lcrypto

all:
		$(CC) $(CFLAGS) Lab3.c -o Lab3
exec:
		./Lab3
clean:
		rm -rf Lab3

test:
		@$(CC) $(CFLAGS) Lab1.c -o Lab3
		@echo "\nRunning Program...\n"
		@./Lab3
```

Fig. 1c. Makefile with necessary flags to utilize bignums on my host machine.

Running "make test" in the terminal compiles and runs the program. (Note: To run this program on other machines the CFLAGS locations will likely need to be modified to the machine its run on.)

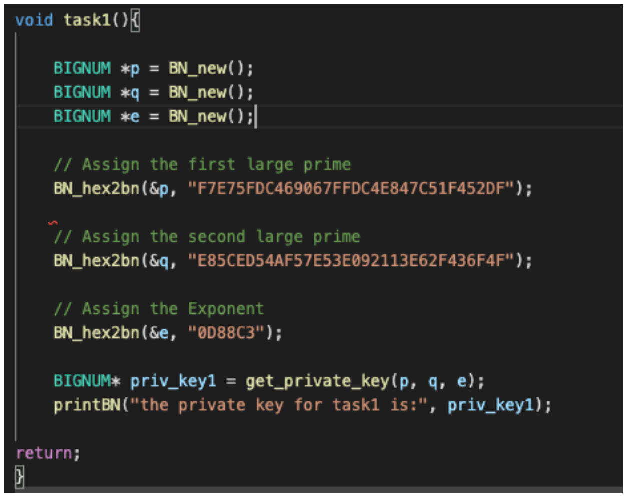
III. By-Step Instructions - Lab Task

A. *Task 1: Deriving the Private Key*

For task 1, I was given three prime numbers :

- p = F7E75FDC469067FFDC4E847C51F452DF
- q = E85CED54AF57E53E092113E62F436F4F
- e = 0D88C3

I then let (e, n) be the public key where n = p*q.

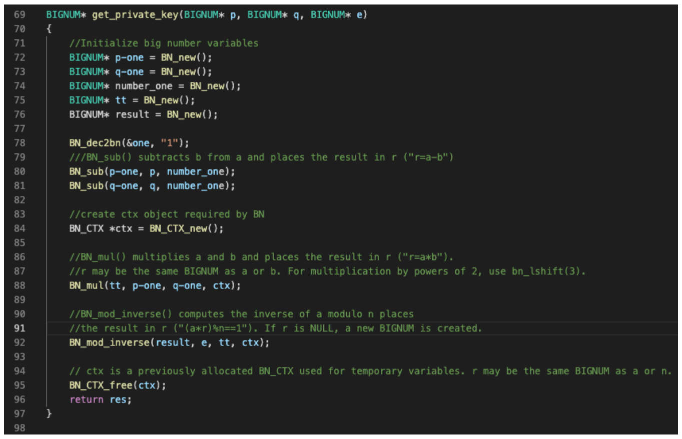


```c
void task1(){

    BIGNUM *p = BN_new();
    BIGNUM *q = BN_new();
    BIGNUM *e = BN_new();

    // Assign the first large prime
    BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");

    // Assign the second large prime
    BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");

    // Assign the Exponent
    BN_hex2bn(&e, "0D88C3");

    BIGNUM* priv_key1 = get_private_key(p, q, e);
    printBN("the private key for task1 is:", priv_key1);

return;
}
```

Fig. 1. Task 1 with setting up provided initial values

Task1() then passes in the given values to the below get_private_key function to compute the key.

Manuscript received April 11, 2022.



```c
BIGNUM* get_private_key(BIGNUM* p, BIGNUM* q, BIGNUM* e)
{
    //Initialize big number variables
    BIGNUM* p-one = BN_new();
    BIGNUM* q-one = BN_new();
    BIGNUM* number_one = BN_new();
    BIGNUM* tt = BN_new();
    BIGNUM* result = BN_new();

    BN_dec2bn(&one, "1");
    ///BN_sub() subtracts b from a and places the result in r ("r=a-b")
    BN_sub(p-one, p, number_one);
    BN_sub(q-one, q, number_one);

    //create ctx object required by BN
    BN_CTX *ctx = BN_CTX_new();

    //BN_mul() multiplies a and b and places the result in r ("r=a*b").
    //r may be the same BIGNUM as a or b. For multiplication by powers of 2, use bn_lshift(3).
    BN_mul(tt, p-one, q-one, ctx);

    //BN_mod_inverse() computes the inverse of a modulo n places
    //the result in r ("(a*r)%n==1"). If r is NULL, a new BIGNUM is created.
    BN_mod_inverse(result, e, tt, ctx);

    // ctx is a previously allocated BN_CTX used for temporary variables. r may be the same BIGNUM as a or n.
    BN_CTX_free(ctx);
    return res;
}
```

Fig. 2. Getprivatekey function called by task 1after initial values entered

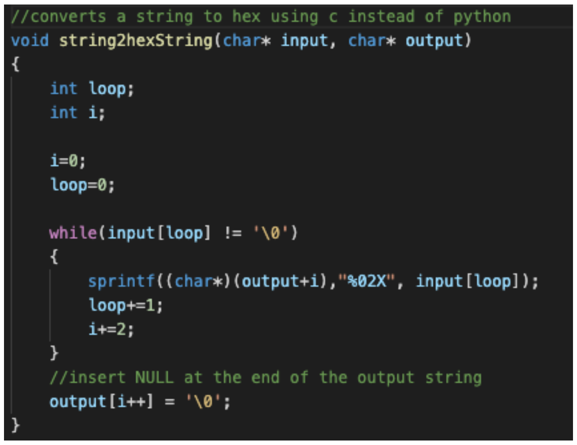
At the beginning of the get_private_key function it initializes the BIGNUM variables to be utilized. Standard arithmetic can't be used on big numbers since it is not a native data type to the c language. This requires additional variables be created such as: one, p-one, and q-one, rather than being able to do: p= p-1. After initialization one is subtracted from both p and q using the BN_sub function. A CTX object is then created for the following calculations. P and Q are multiplied and then the inverse is taken between e and the result of the multiplication between p and q. The returning result is the private key.

```
the private key for task1 is:
3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28
A9B496AEB
```

Fig. 3. Resulting Generated key

B. *Task 2: Encrypting a Message*

For Task 2 I was asked to encrypt a message using RSA. The message I was told to encrypt was my name + my cwid. I began by writing the c code to convert the string to hexadecimal format rather than use the python command.



```c
//converts a string to hex using c instead of python
void string2hexString(char* input, char* output)
{
    int loop;
    int i;

    i=0;
    loop=0;

    while(input[loop] != '\0')
    {
        sprintf((char*)(output+i),"%02X", input[loop]);
        loop+=1;
        i+=2;
    }
    //insert NULL at the end of the output string
    output[i++] = '\0';
}
```

Fig. 4. Convert string to hex[2]

The plaintext value in hex for my string was:
5461796C6F724164616D732B3131373032353530

I was then given the following variables to use in the encryption:

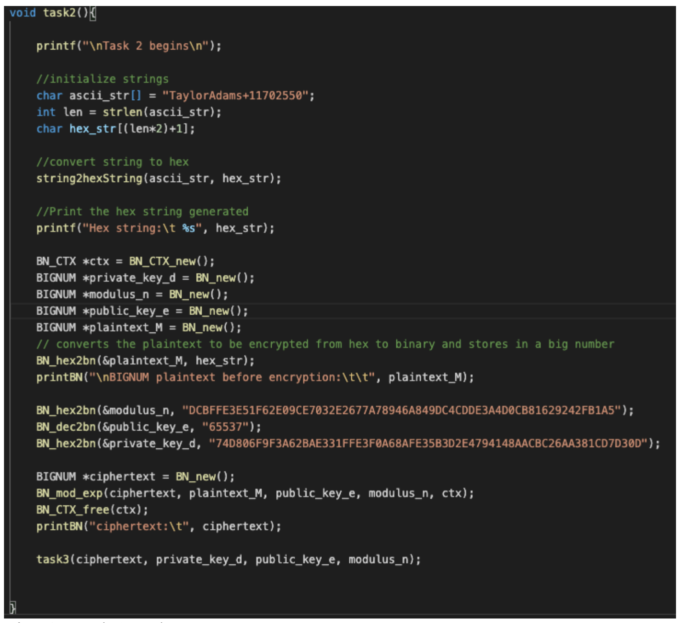
- modulus_n
  - DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
- public_key_e
  - 010001
- Private_key_d
  - 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D

I then initialized the BIGNUM variables. I assigned those variables the values provided earlier. Following these two actions I created a BIGNUM ciphertext variable to store the generated ciphertext. I utilized the BN_mod_exp function to generate the ciphertext using the provided values.
BN_mod_exp(ciphertext, plaintext_M, public_key_e, modulus_n, ctx);

The resulting ciphertext was:

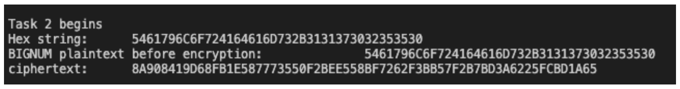
ciphertext:
8A908419D68FB1E587773550F2BEE558BF7262F3BB57F2B7BD3A6225FCBD1A65

Fig. 5. Task 2 resulting ciphertext



```
void task2(){

    printf("\nTask 2 begins\n");

    //initialize strings
    char ascii_str[] = "TaylorAdams+11702550";
    int len = strlen(ascii_str);
    char hex_str[(len*2)+1];

    //convert string to hex
    string2hexString(ascii_str, hex_str);

    //Print the hex string generated
    printf("Hex string:\t %s", hex_str);

    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *private_key_d = BN_new();
    BIGNUM *modulus_n = BN_new();
    BIGNUM *public_key_e = BN_new();
    BIGNUM *plaintext_M = BN_new();
    // converts the plaintext to be encrypted from hex to binary and stores in a big number
    BN_hex2bn(&plaintext_M, hex_str);
    printBN("\nBIGNUM plaintext before encryption:\t\t", plaintext_M);

    BN_hex2bn(&modulus_n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_dec2bn(&public_key_e, "65537");
    BN_hex2bn(&private_key_d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

    BIGNUM *ciphertext = BN_new();
    BN_mod_exp(ciphertext, plaintext_M, public_key_e, modulus_n, ctx);
    BN_CTX_free(ctx);
    printBN("ciphertext:\t", ciphertext);

    task3(ciphertext, private_key_d, public_key_e, modulus_n);


}
```

Fig. 6. Task 2 code



```
Task 2 begins
Hex string:      5461796C6F724164616D732B3131373032353530
BIGNUM plaintext before encryption:             5461796C6F724164616D732B3131373032353530
ciphertext:      8A908419D68FB1E587773550F2BEE558BF7262F3BB57F2B7BD3A6225FCBD1A65
```

Fig. 7. Task 2 resulting outputs

C. *Task 3: RSA decryption*

For this task I was required to decrypt a ciphertext hex string and convert it back to ascii to reveal the message. The public and private keys were the same ones as in task 2 so task 3 was called from task 2 passing in the appropriate values and BIGNUM variables, so that I would not need to recreate them.

A new ciphertext was however provided by the lab to decrypt. I replaced the BIGNUM value of ciphertext with the ciphertext provided by the lab:

C = 8C0F971DF2F3672B28811407E2DABB E1DA0FEBBBDFC7DCB67396567EA1E2493F

Fig. 8. Ciphertext provided by the lab

After changing the ciphertext value I ran the same function as before BN_mod_exp();

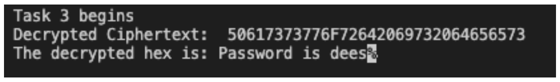
BN_mod_exp(decrypted_ciphertext,ciphertext, private_key_d, modulus_n, ctx);

Fig. 9. Task3 BN_mod_exp() call;

The above function take in an empty BIGNUM decrypted_ciphertext as the output, the ciphertext itself, the private key, the modulus, and the BIGNUM CTX var.

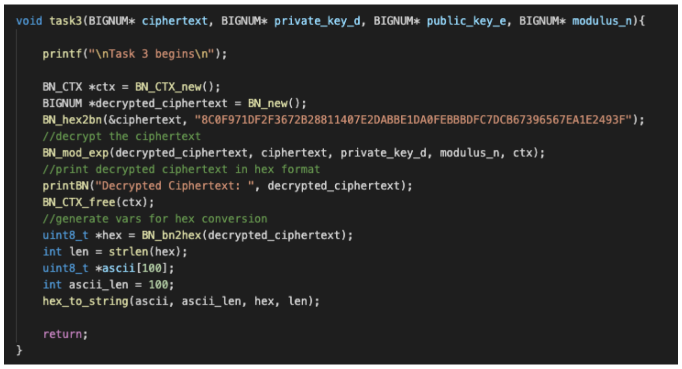
I used the BN_bn2hex method call on the decrypted_ciphertext variable to convert the var to a hex format for converting to an ascii string.
I then found some C code online to convert from hex to ascii characters. This function is called hex_to_string(). The resulting output after conversion to ascii was:

Password is dees



```
Task 3 begins
Decrypted Ciphertext:  50617373776F72642069732064656573
The decrypted hex is: Password is dees%
```

Fig. 10. Task 3 resulting plaintext from the encrypted ciphertext



```
void task3(BIGNUM* ciphertext, BIGNUM* private_key_d, BIGNUM* public_key_e, BIGNUM* modulus_n){

    printf("\nTask 3 begins\n");

    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *decrypted_ciphertext = BN_new();
    BN_hex2bn(&ciphertext, "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");
    //decrypt the ciphertext
    BN_mod_exp(decrypted_ciphertext, ciphertext, private_key_d, modulus_n, ctx);
    //print decrypted ciphertext in hex format
    printBN("Decrypted Ciphertext: ", decrypted_ciphertext);
    BN_CTX_free(ctx);
    //generate vars for hex conversion
    uint8_t *hex = BN_bn2hex(decrypted_ciphertext);
    int len = strlen(hex);
    uint8_t *ascii[100];
    int ascii_len = 100;
    hex_to_string(ascii, ascii_len, hex, len);

    return;
}
```

Fig. 11. Task 3 code

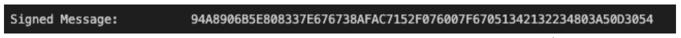
D. *Task 4: Signing a Message*

In task 4, I was asked to sign a message using RSA. The public and private keys were the same from task 2. But the message to sign was "Taylor Adams owes you $2000." I was also asked to sign another string as well "Taylor Adams owes you $3000." and compare the resulting signed values.

To begin this task I declared my two strings, got the length and passed them into the method used earlier to convert them to hexadecimal format (string2hexString()). Once I had my hex strings I created BIGNUM variables for both of them. I then Initialized BIGNUM variables for the ctx, private key, modulus, and public key. I then ran BN_mod_exp on both of the stings and compared the digital signatures.

Manuscript received April 11, 2022.

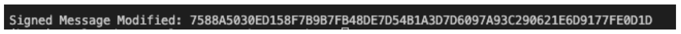
The first message “Taylor Adams owes you $2000.” Became the signed message:



```
Signed Message:         94A8906B5E808337E676738AFAC7152F076007F67051342132234803A50D3054
```

Fig. 12. Signed message from string “Taylor Adams owes you $2000.

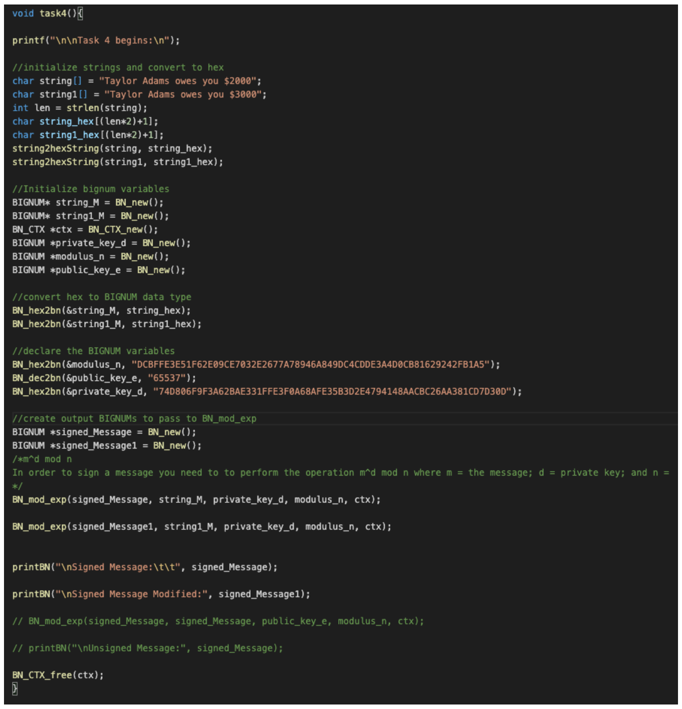
The second message “Taylor Adams owes you $3000” Became the signed message:



```
Signed Message Modified: 7588A5030ED158F7B9B7FB48DE7D54B1A3D7D6097A93C290621E6D9177FE0D1D
```

Fig. 13. Signed message from string “Taylor Adams owes you $3000.

It is clearly visible that both of this hex strings are completely different with just one char modified. This displays the non-repudiation where the sender cannot deny they sent a message when it is decrypted with their private key. This also shows how an attacker would need to have access to a victims private key in order to sign a fake message and make it believable because an attacker would not be able change the value of a character in the signed message to change the resulting decrypted message from $2000 to $3000.



```c
void task4(){

printf("\n\nTask 4 begins:\n");

//initialize strings and convert to hex
char string[] = "Taylor Adams owes you $2000";
char string1[] = "Taylor Adams owes you $3000";
int len = strlen(string);
char string_hex[(len*2)+1];
char string1_hex[(len*2)+1];
string2hexString(string, string_hex);
string2hexString(string1, string1_hex);

//Initialize bignum variables
BIGNUM* string_M = BN_new();
BIGNUM* string1_M = BN_new();
BN_CTX *ctx = BN_CTX_new();
BIGNUM *private_key_d = BN_new();
BIGNUM *modulus_n = BN_new();
BIGNUM *public_key_e = BN_new();

//convert hex to BIGNUM data type
BN_hex2bn(&string_M, string_hex);
BN_hex2bn(&string1_M, string1_hex);

//declare the BIGNUM variables
BN_hex2bn(&modulus_n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
BN_dec2bn(&public_key_e, "65537");
BN_hex2bn(&private_key_d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

//create output BIGNUMs to pass to BN_mod_exp
BIGNUM *signed_Message = BN_new();
BIGNUM *signed_Message1 = BN_new();
/*m^d mod n
In order to sign a message you need to to perform the operation m^d mod n where m = the message; d = private key; and n =
*/
BN_mod_exp(signed_Message, string_M, private_key_d, modulus_n, ctx);

BN_mod_exp(signed_Message1, string1_M, private_key_d, modulus_n, ctx);


printBN("\nSigned Message:\t\t", signed_Message);

printBN("\nSigned Message Modified:", signed_Message1);

// BN_mod_exp(signed_Message, signed_Message, public_key_e, modulus_n, ctx);

// printBN("\nUnsigned Message:", signed_Message);

BN_CTX_free(ctx);
}
```

Fig. 14. Task 4 code

E. *Task 5: Verifying a Signature*

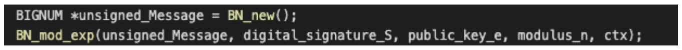
In task 5 I was asked to verify an RSA signature. I know the resulting plaintext is “Launch a missile.” And the public key signature’s values:

- Digital Signature:
  - 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
- Public key:
  - 010001 (hex: 65537)
- Modulus:
  - AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115

Manuscript received April 11, 2022.

This lab also asks us to corrupt the text and test what the resulting value would be.

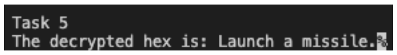
As before, I began by initializing the BIGNUM variables for the verification. I then inserted the values provided by the lab into the BIGNUM variables. I then utilized the BN_mod_exp function as before to verify the signature of the sender.



```c
BIGNUM *unsigned_Message = BN_new();
BN_mod_exp(unsigned_Message, digital_signature_S, public_key_e, modulus_n, ctx);
```

Fig. 15. Task 5 BN_mod_exp

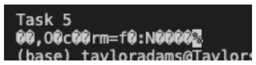
With the uncorrupted digital signature, I got the expected output of “Launch a missile”)



```
Task 5
The decrypted hex is: Launch a missile.%
```

Fig. 16. Task 5 BN_mod_exp

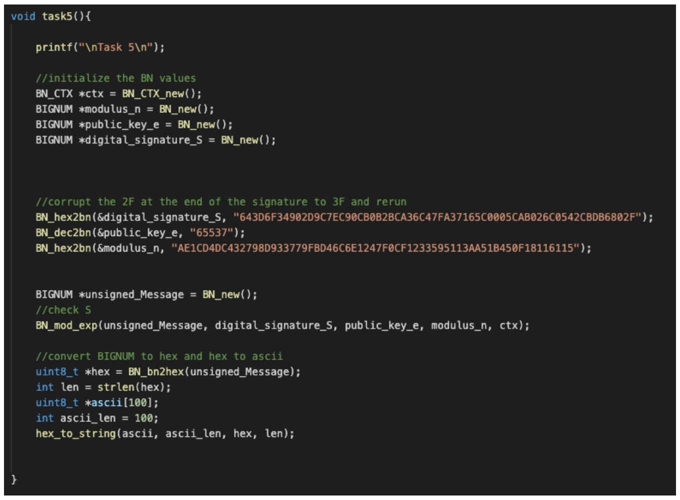
However, when I change the end of the digital signature from 2F to 3F and convert to ascii I get:



```
Task 5
��,0�c��rm=f�:N����%
(base) tayloradams@Taylor
```

Fig. 17. Task 5 BN_mod_exp

Task 5 displays how an attacker cannot modify a digital signature as it will likely just end up corrupting itself and therefore be unreadable.



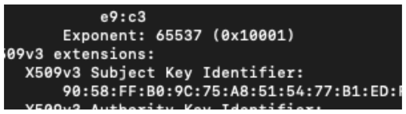
```c
void task5(){

    printf("\nTask 5\n");

    //initialize the BN values
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *modulus_n = BN_new();
    BIGNUM *public_key_e = BN_new();
    BIGNUM *digital_signature_S = BN_new();



    //corrupt the 2F at the end of the signature to 3F and rerun
    BN_hex2bn(&digital_signature_S, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");
    BN_dec2bn(&public_key_e, "65537");
    BN_hex2bn(&modulus_n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");


    BIGNUM *unsigned_Message = BN_new();
    //check S
    BN_mod_exp(unsigned_Message, digital_signature_S, public_key_e, modulus_n, ctx);

    //convert BIGNUM to hex and hex to ascii
    uint8_t *hex = BN_bn2hex(unsigned_Message);
    int len = strlen(hex);
    uint8_t *ascii[100];
    int ascii_len = 100;
    hex_to_string(ascii, ascii_len, hex, len);


}
```

Fig. 18. Task 5 code

F. *Task 6: Manually Verifying an X.509 Certificate*

For this task I chose to verify the X.509 certificate from www.redcross.org. I began this task by creating a separate folder for task 6 and then creating both c0.pem and c1.pem files. I then ran the command: “`openssl s_client -connect www.example.org:443 -showcerts`” and copied both certificates into their respective .pem files. In order to get the modulus I ran the command: “`openssl x509 -in c1.pem -noout -modulus`”. I then initialized a BIGNUM modulus_n in my task6 code and copied the modulus into the initialized BIGNUM.

Following the instructions I ran the command: “openssl x509 -in c1.pem -text -noout” to get the public key.

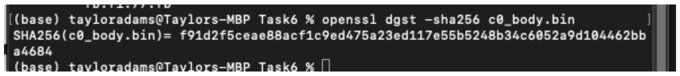


```
            e9:c3
        Exponent: 65537 (0x10001)
09v3 extensions:
    X509v3 Subject Key Identifier:
        90:58:FF:B0:9C:75:A8:51:54:77:B1:ED:
    X509v3 Authority Key Identifier:
```

Fig. 19. Results from running openssl x509 -in c1.pem -text -noout

After getting the public key, I inserted it into my program the same as I did for the modulus. I then ran the command: "`openssl x509 -in c0.pem -text -noout`" to get the digital signature, saved it into a text file and ran the following command on the text file to remove the extra spaces and colons: "`cat signature | tr -d '[:space:]:'`".

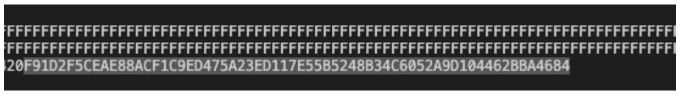
Following the instructions further I ran "`openssl asn1parse -i -in c0.pem -strparse 4 -out c0_body.bin -noout`" to generate a .bin file of the c0.pem file. Following this command, I ran the command: "`openssl dgst -sha256 c0_body.bin`" which gave me the message digest:



```
(base) tayloradams@Taylors-MBP Task6 % openssl dgst -sha256 c0_body.bin
SHA256(c0_body.bin)= f91d2f5ceae88acf1c9ed475a23ed117e55b5248b34c6052a9d104462bb
a4684
(base) tayloradams@Taylors-MBP Task6 %
```

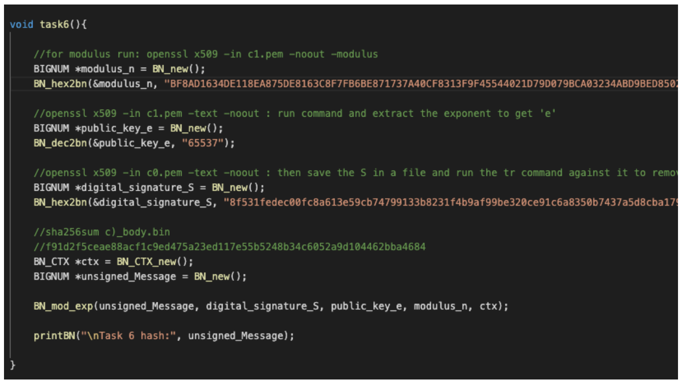
Fig. 20. Output of command: openssl dgst -sha256 c0_body.bin

I then ran my task6() code and was able to generate the exact same hash albeit with padding attached.



```
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
20F91D2F5CEAE88ACF1C9ED475A23ED117E55B5248B34C6052A9D104462BBA4684
```

Fig. 21. Task 6 Output



```
void task6(){

    //for modulus run: openssl x509 -in c1.pem -noout -modulus
    BIGNUM *modulus_n = BN_new();
    BN_hex2bn(&modulus_n, "BF8AD1634DE118EA875DE8163C8F7FB6BE871737A40CF8313F9F45544021D79D079BCA03234ABD9BED850

    //openssl x509 -in c1.pem -text -noout : run command and extract the exponent to get 'e'
    BIGNUM *public_key_e = BN_new();
    BN_dec2bn(&public_key_e, "65537");

    //openssl x509 -in c0.pem -text -noout : then save the S in a file and run the tr command against it to remo
    BIGNUM *digital_signature_S = BN_new();
    BN_hex2bn(&digital_signature_S, "8f531fedec00fc8a613e59cb74799133b8231f4b9af99be320ce91c6a8350b7437a5d8cba17

    //sha256sum c)_body.bin
    //f91d2f5ceae88acf1c9ed475a23ed117e55b5248b34c6052a9d104462bba4684
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *unsigned_Message = BN_new();

    BN_mod_exp(unsigned_Message, digital_signature_S, public_key_e, modulus_n, ctx);

    printBN("\nTask 6 hash:", unsigned_Message);

}
```

Fig. 22. Task 6 Code

IV. Discussions

This lab has allowed me to gain a much greater understanding of not only RSA encryption, but also the general concept of public key cryptography. This lab was the first time I have had the opportunity to sit down and learn the fundamentals of asymmetric encryption. Through the use of openssl and the bignums library, I gained a greater understanding of installing and utilizing this library along with public key encryption.

Lab Difficulties

This lab did have its challenges. Installing and utilizing the bignums library on m host machine took an hour or so to figure out. I ended up having to google several errors I had when attempting to include the bignums library in my C program. I found that my install had not put openssl in the apporate system library on install and I needed to add the additional flags of: '-lssl' and '-lcrypto' for the C program to compile with the bignums library included.

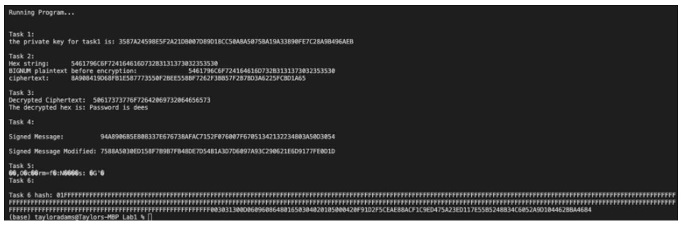
My main difficulty for the lab was task 1. I found the online bignums documentation difficult to follow. This made it challenging to understand the functions needed to create the private key for task 1.

Another difficulty for the entire lab was having to adjust to utilizing the Bignums data type since regular arithmetic expressions would not work with the bignums datatype because of its size. This made reading and understanding the code slightly more challenging than if I had been able to utilize integers or longs.

The remaining task were not as difficult as the first one was. However, I did have to redo task 6 with a different website because the first time I tried to verify the certificate I ended up with an output that didn't make sense.

V. Conclusion

I was able to learn a lot from this lab and correlate the information learned in class with real world applications. By completing this lab, I now have a much better grasp of RSA encryption and how it can be used to verify the authenticity, integrity and non-repudiation of a message. This lab along with our textbook should allow me to gain a foothold on cryptography and the challenges of its implementation.



```
Running Program...


Task 1:
the private key for task1 is: 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB

Task 2:
Hex string:      5461796C6F724164616D732B3131373032353530
BIGNUM plaintext before encryption:             5461796C6F724164616D732B3131373032353530
ciphertext:      8A908419D68FB1E587773550F2BEE558BF7262F38B57F2B7BD3A6225FCBD1A65

Task 3:
Decrypted Ciphertext:  50617373776F726420697320646565573
The decrypted hex is: Password is dees

Task 4:

Signed Message:         94A8906B5E808337E676738AFAC7152F076007F670513421322348O3A50D3054

Signed Message Modified: 7588A5030ED158F7B9B7FB48DE7D54B1A3D7D6097A93C290621E6D9177FE0D1D

Task 5:
00,00c00rm=f0:N0000s: 0G'0
Task 6:

Task 6 hash: 01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF003031300D060960864801650304020105000420F91D2F5CEAE88ACF1C9ED475A23ED117E55B5248B34C6052A9D104462BBA4684
(base) tayloradams@Taylors-MBP Lab1 %
```

Fig. 23. Lab entire program output for all task

References

[1]"American Red Cross." https://www.redcross.org (accessed Mar. 05, 2022).

[2]"Convert ASCII string (char[]) to hexadecimal string (char[]) in C." https://www.includehelp.com/c/convert-ascii-string-to-hexadecimal-string-in-c.aspx (accessed Mar. 05, 2022).

[3]"Convert Hexadecimal value String to ASCII value String," GeeksforGeeks, May 22, 2018. https://www.geeksforgeeks.org/convert-hexadecimal-value-string-ascii-value-string/ (accessed Mar. 05, 2022).

[4]"SEED Project." https://seedsecuritylabs.org/Labs_20.04/Crypto/Crypto_RSA/ (accessed Mar. 05, 2022).

Manuscript received April 11, 2022.