Hannah McLaughlin

RSA Public-Key Encryption and Signature Lab

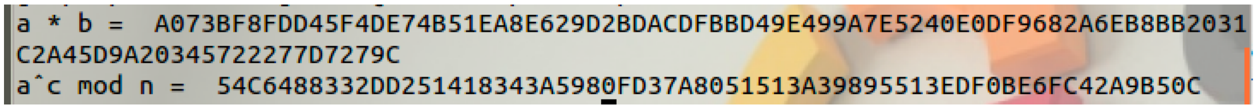
CIS5370 – Computer Security

Introduction

The goal of this lab is to gain hands-on experiences with the RSA algorithm. The RSA algorithm involves the computations on large numbers. Since our operators can only operate on primitive data types, such as 32-bit integers or 64-bit long integers, we must use a library that can perform arithmetic operations on integers of arbitrary size. In this lab, I used the Big Number library provided by `openssl`. I installed this by doing:

```
$ sudo apt-get update
$ sudo apt-get install libssl-dev
```

I compiled the program, `bn_sample.c`, given in the project description. I ran `$ gcc bn_sample.c -lcrypto` to compile the program using the crypto library. The following was the result from running the program.

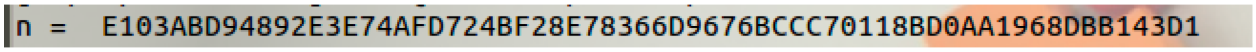


```
a * b =  A073BF8FDD45F4DE74B51EA8E629D2BDACDFBBD49E499A7E5240E0DF9682A6EB8BB2031
C2A45D9A20345722277D7279C
a^c mod n =  54C6488332DD251418343A5980FD37A8051513A39895513EDF0BE6FC42A9B50C
```

Task 1: Deriving the Private Key

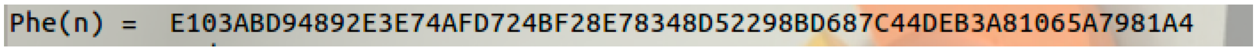
```
p = F7E75FDC469067FFDC4E847C51F452DF
q = E85CED54AF57E53E092113E62F436F4F
e = 0D88C3
```

First, I calculated `n` by multiplying `p * q.`



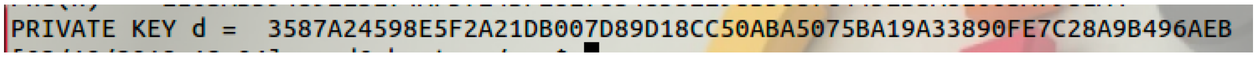
```
n =  E103ABD94892E3E74AFD724BF28E78366D9676BCCC70118BD0AA1968DBB143D1
```

To find ϕ(`n`) or Phe(`n`), I calculated `(p – 1)*(q – 1)`.



```
Phe(n) =  E103ABD94892E3E74AFD724BF28E78348D52298BD687C44DEB3A81065A7981A4
```

With the given `e`, I was able to use the modulo inverse function provided to us in the description to find private key `d`. When I compiled and ran the program, I found private key `d.`



```
PRIVATE KEY d =  3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
```

(see attached code – Task1.c)

Task 2: Encrypting a Message

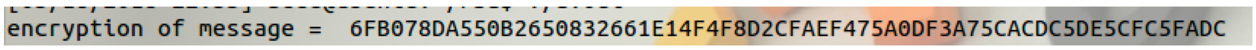
In this task, we were given the public key (n, e) and told to encrypt the message, "A top secret!" First, I converted the message to a hex string using the following python command with the result:

```
$ python -c 'print("A top secret!".encode("hex"))'
```

Hannah McLaughlin

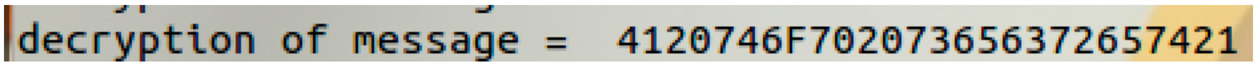
`4120746f702073656372657421`

The message needed to be a hexadecimal, so that I could perform calculations on it. To encrypt a message, the ciphertext is calculated using the equation `ciphertext = message^e mod n`. I used the `BN_mod_exp()` function to easily calculate this.



```
encryption of message =  6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
```

Since I was also provided the private key, I implemented the decryption as well, just ensure sure I got the original message back. The decryption equation is `message = ciphertext^d mod n.` I did, in fact, get the original hexadecimal message back.

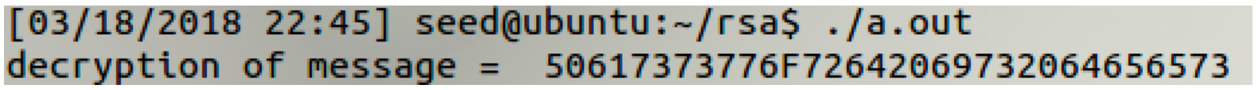


```
decryption of message =  4120746F702073656372657421
```

(see attached code - Task2.c)

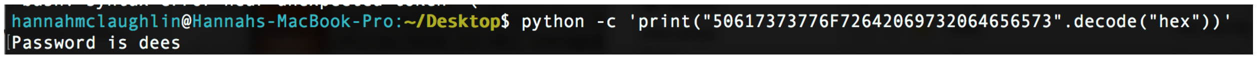
Task 3: Decrypting a Message

Since I had previously implemented the decryption equation, I used this code to decrypt the ciphertext provided.



```
[03/18/2018 22:45] seed@ubuntu:~/rsa$ ./a.out
decryption of message =  50617373776F72642069732064656573
```

Next I ran a python command to convert the hex string back to a plain ASCII string. I was having trouble running python on the SEED VM, so I used my host.

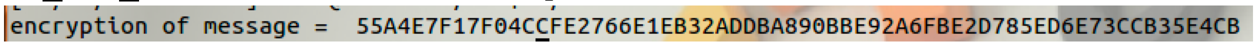


```
hannahmclaughlin@Hannahs-MacBook-Pro:~/Desktop$ python -c 'print("50617373776F72642069732064656573".decode("hex"))'
Password is dees
```

The encrypted message is “Password is dees”
(see attached code – Task3.c)

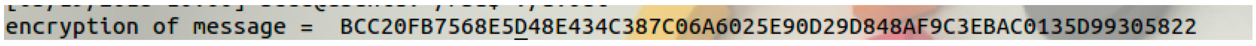
Task 4: Signing a Message

Previously in the RSA encryption tasks, I was using the public key to encrypt. However, in digital signatures, you encrypt with the private key and the receiver compares using the public key. The equation to sign a message is `Message^d mod n.` Again, I began by converting the message to hexadecimal, that way I can operate on the string. I used the python script:
`$ python -c ‘print("I owe you $2000.".encode("hex"))’`Next, I used the `BN_mod_exp()` function to create the ciphertext.



```
encryption of message =  55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
```

I changed the message to say “$3000” instead of “$2000”. Again, I converted the message to hexadecimal and ran the digital signature equation on it.

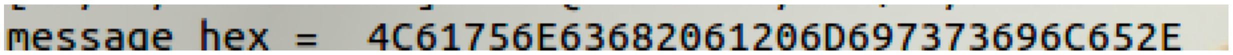


```
encryption of message =  BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
```

Hannah McLaughlin

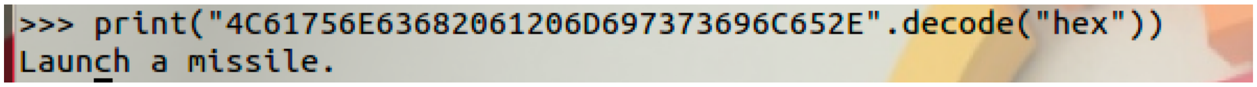
The original hexadecimal message encodings only differed by one byte. However, the digital signatures of the two messages are extremely different. This is a protection of the modulus operation.
(see attached code – Task4.c)

Task 5: Verifying a Signature
To verify that a signature is the correct message, I used the following formula to get the hexadecimal representation of the message: `Signature^e mod n.` My program returned the hex string



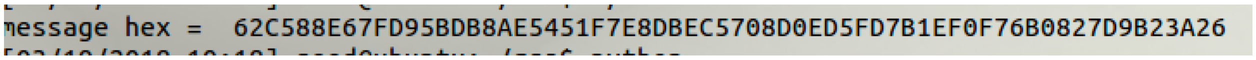
```
message hex =  4C61756E63682061206D697373696C652E
```

I decoded this string using python again and saw that I received the original message.



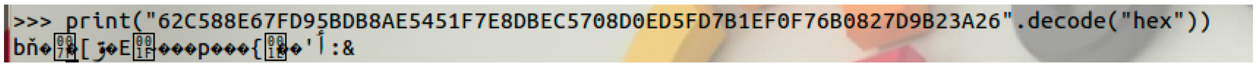
```
>>> print("4C61756E63682061206D697373696C652E".decode("hex"))
Launch a missile.
```

Next, I modified the signature from 2F to 3F and ran the program again. I got an entirely different hex string.



```
message hex =  62C588E67FD95BDB8AE5451F7E8DBEC5708D0ED5FD7B1EF0F76B0827D9B23A26
```

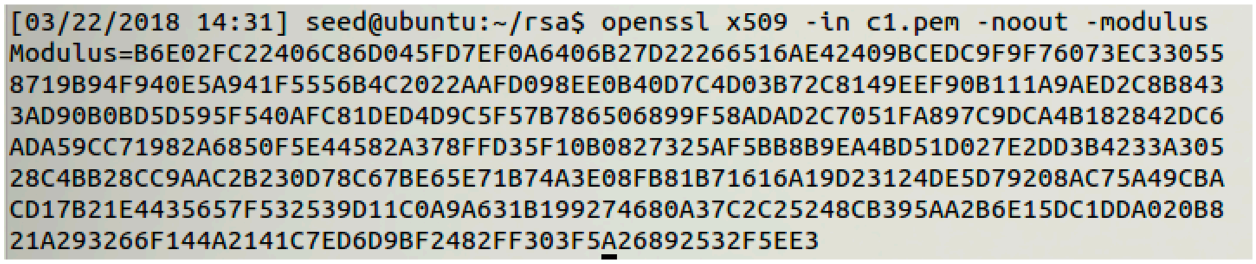
Just for fun, I tried decoding the message and got gibberish.



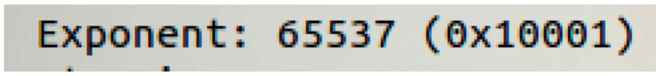
```
>>> print("62C588E67FD95BDB8AE5451F7E8DBEC5708D0ED5FD7B1EF0F76B0827D9B23A26".decode("hex"))
bň◆[ǯ◆E◆◆◆p◆◆◆{◆'|:&
```

The reason I got an entirely different hex string is because of simple math. When I raised the signature to the same exponent, the base is changed so the result is different.

Task 6: Manually Verifying an X.509 Certificate
I downloaded the certificate from Facebook.com using the command `openssl s_client -connect www.facebook.org:443 -showcerts.` I got two certificates and saved them to the c0.pem and c1.pem. Next, I extracted the public key (n, e) from the issuer's certificate.



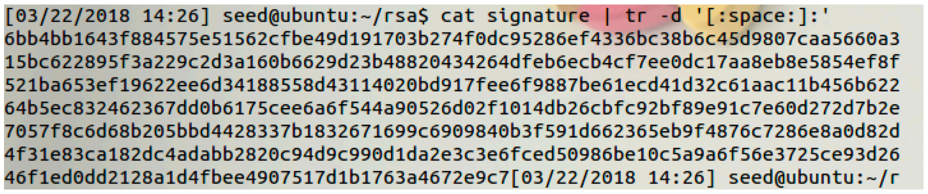
```
[03/22/2018 14:31] seed@ubuntu:~/rsa$ openssl x509 -in c1.pem -noout -modulus
Modulus=B6E02FC22406C86D045FD7EF0A6406B27D22266516AE42409BCEDC9F9F76073EC33055
8719B94F940E5A941F5556B4C2022AAFD098EE0B40D7C4D03B72C8149EEF90B111A9AED2C8B843
3AD90B0BD5D595F540AFC81DED4D9C5F57B786506899F58ADAD2C7051FA897C9DCA4B182842DC6
ADA59CC71982A6850F5E44582A378FFD35F10B0827325AF5BB8B9EA4BD51D027E2DD3B4233A305
28C4BB28CC9AAC2B230D78C67BE65E71B74A3E08FB81B71616A19D23124DE5D79208AC75A49CBA
CD17B21E4435657F532539D11C0A9A631B199274680A37C2C25248CB395AA2B6E15DC1DDA020B8
21A293266F144A2141C7ED6D9BF2482FF303F5A26892532F5EE3
```



```
Exponent: 65537 (0x10001)
```

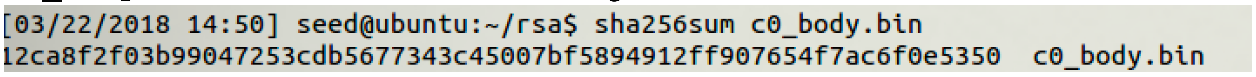
To extract the signature from the server's certificate, I ran `$ openssl x509 -in c0.pem -text -noout` and then I stripped it of the space and colons.

Hannah McLaughlin



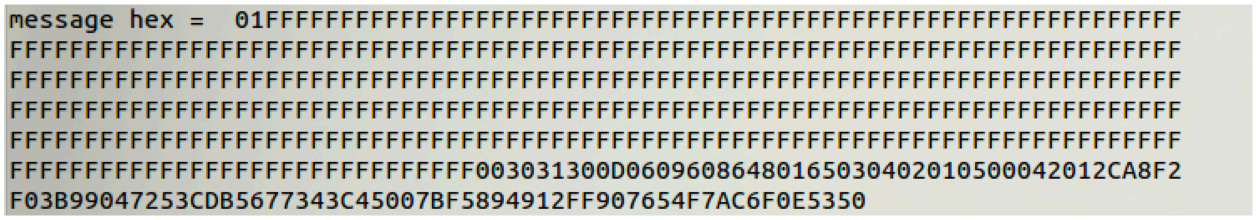
```
[03/22/2018 14:26] seed@ubuntu:~/rsa$ cat signature | tr -d '[:space:]:'
6bb4bb1643f884575e51562cfbe49d191703b274f0dc95286ef4336bc38b6c45d9807caa5660a3
15bc622895f3a229c2d3a160b6629d23b48820434264dfeb6ecb4cf7ee0dc17aa8eb8e5854ef8f
521ba653ef19622ee6d34188558d43114020bd917fee6f9887be61ecd41d32c61aac11b456b622
64b5ec832462367dd0b6175cee6a6f544a90526d02f1014db26cbfc92bf89e91c7e60d272d7b2e
7057f8c6d68b205bbd4428337b1832671699c6909840b3f591d662365eb9f4876c7286e8a0d82d
4f31e83ca182dc4adabb2820c94d9c990d1da2e3c3e6fced50986be10c5a9a6f56e3725ce93d26
46f1ed0dd2128a1d4fbee4907517d1b1763a4672e9c7[03/22/2018 14:26] seed@ubuntu:~/r
```

Next, I extracted the body of the server's certificate which are encoded using the abstract Syntax Notation.One standard. I used Openssl's `asn1parse` command and saved it to a file named `c0_body.bin.` Thenm I hashed the file using SHA256.



```
[03/22/2018 14:50] seed@ubuntu:~/rsa$ sha256sum c0_body.bin
12ca8f2f03b99047253cdb5677343c45007bf5894912ff907654f7ac6f0e5350  c0_body.bin
```

To verify the signature, I used the same code from Task 5. I changed the public key and the signature and ran the program using the `-lcrypt` flag. Even though, the hash looks different from the verified signature, you can see at the end, that they are the same.



```
message hex =  01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF003031300D06096086480165030402010500042012CA8F2
F03B99047253CDB5677343C45007BF5894912FF907654F7AC6F0E5350
```

(see Task6.c)