CSCI 476: Lab 11

Nathan Stouffer

April 29, 2020

Task 1

Task 1 asks for the private key to be computed given the following information:

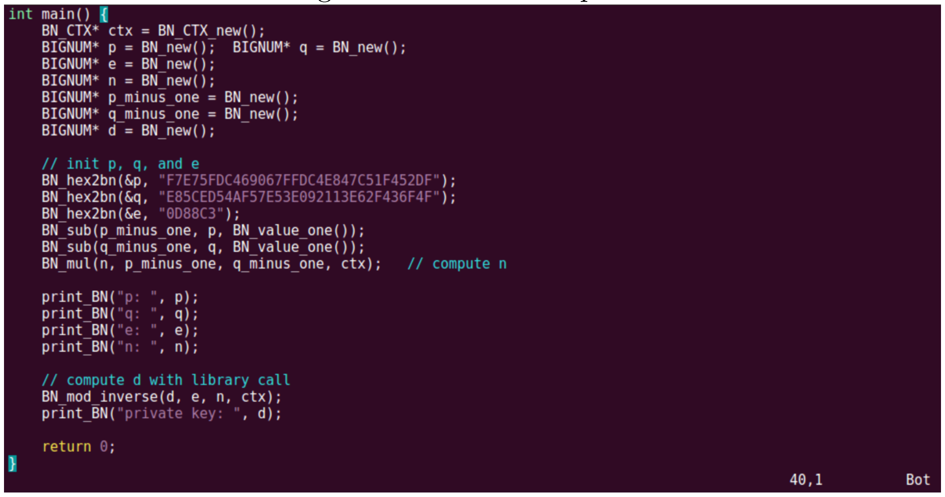
1. p = F7E75FDC469067FFDC4E847C51F452DF

2. q = E85CED54AF57E53E092113E62F436F4F

3. e = 0D88C3

The private key d is computed using the command BN_mod_inverse() from the openssl big number library. The code to do this is shown in Figure 1.

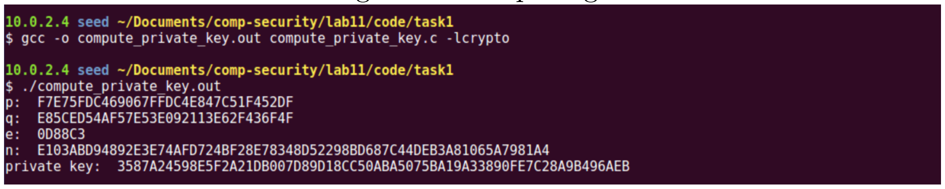
Figure 1: Code to compute d



```
int main() {
    BN_CTX* ctx = BN_CTX_new();
    BIGNUM* p = BN_new();  BIGNUM* q = BN_new();
    BIGNUM* e = BN_new();
    BIGNUM* n = BN_new();
    BIGNUM* p_minus_one = BN_new();
    BIGNUM* q_minus_one = BN_new();
    BIGNUM* d = BN_new();

    // init p, q, and e
    BN_hex2bn(&p, "F7E75FDC469067FFDC4E847C51F452DF");
    BN_hex2bn(&q, "E85CED54AF57E53E092113E62F436F4F");
    BN_hex2bn(&e, "0D88C3");
    BN_sub(p_minus_one, p, BN_value_one());
    BN_sub(q_minus_one, q, BN_value_one());
    BN_mul(n, p_minus_one, q_minus_one, ctx);   // compute n

    print_BN("p: ", p);
    print_BN("q: ", q);
    print_BN("e: ", e);
    print_BN("n: ", n);

    // compute d with library call
    BN_mod_inverse(d, e, n, ctx);
    print_BN("private key: ", d);

    return 0;
}
                                                             40,1          Bot
```

Then Figure 2 shows the compilation and output of the program. The private key is printed along with the output.

Figure 2: Computing d



```
10.0.2.4 seed ~/Documents/comp-security/lab11/code/task1
$ gcc -o compute_private_key.out compute_private_key.c -lcrypto

10.0.2.4 seed ~/Documents/comp-security/lab11/code/task1
$ ./compute_private_key.out
p:  F7E75FDC469067FFDC4E847C51F452DF
q:  E85CED54AF57E53E092113E62F436F4F
e:  0D88C3
n:  E103ABD94892E3E74AFD724BF28E78348D52298BD687C44DEB3A81065A7981A4
private key:  3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
```

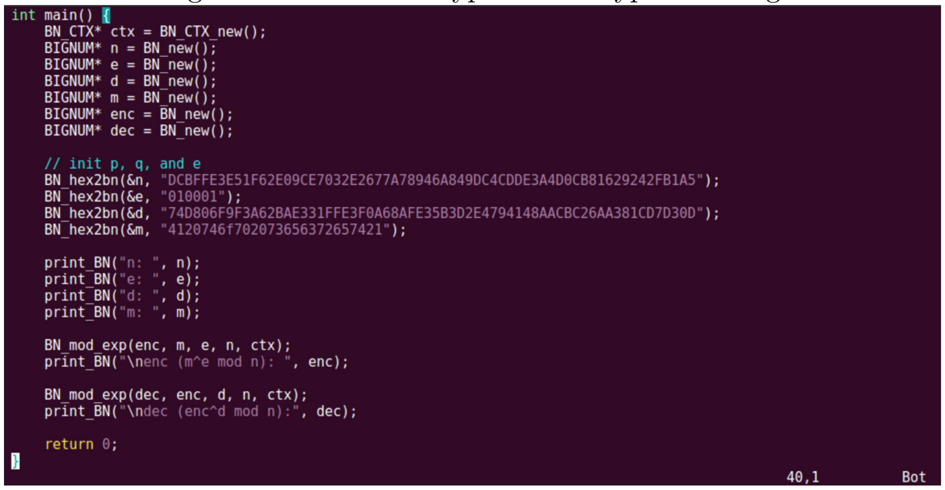
Task 2

Task 2 focuses on the encryption and decryption of a message. The following information is known:

1. m = A top secret!
2. e = 010001 (this hex value equals to decimal 65537)
3. n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
4. d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D

The message m is converted to hex and then ran through the program shown in Figure 3. To encrypt the message, $c = m^d \mod n$ is computed. To decrypt, $m' = c^e \mod n$ is computed. By certain laws of mathematics m and m' must be the same.

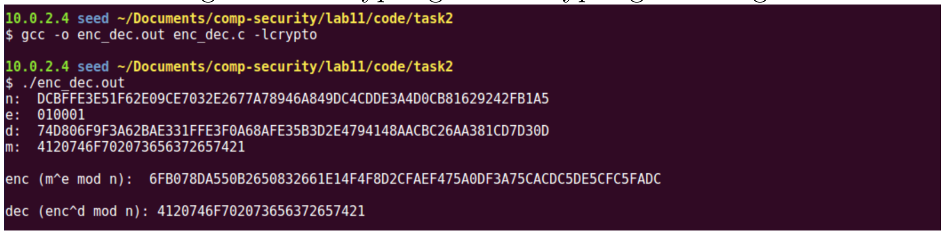
Figure 3: Code to encrypt and decrypt a message



```
int main() {
    BN_CTX* ctx = BN_CTX_new();
    BIGNUM* n = BN_new();
    BIGNUM* e = BN_new();
    BIGNUM* d = BN_new();
    BIGNUM* m = BN_new();
    BIGNUM* enc = BN_new();
    BIGNUM* dec = BN_new();

    // init p, q, and e
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&m, "4120746f702073656372657421");

    print_BN("n: ", n);
    print_BN("e: ", e);
    print_BN("d: ", d);
    print_BN("m: ", m);

    BN_mod_exp(enc, m, e, n, ctx);
    print_BN("\nenc (m^e mod n): ", enc);

    BN_mod_exp(dec, enc, d, n, ctx);
    print_BN("\ndec (enc^d mod n):", dec);

    return 0;
}
                                                                40,1          Bot
```

Figure 4 then shows the output of the process. In fact, the original message m and the decrypted message are the same.

Figure 4: Encrypting and decrypting a message



```
10.0.2.4 seed ~/Documents/comp-security/lab11/code/task2
$ gcc -o enc_dec.out enc_dec.c -lcrypto

10.0.2.4 seed ~/Documents/comp-security/lab11/code/task2
$ ./enc_dec.out
n:  DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e:  010001
d:  74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
m:  4120746F702073656372657421

enc (m^e mod n):  6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC

dec (enc^d mod n): 4120746F702073656372657421
```

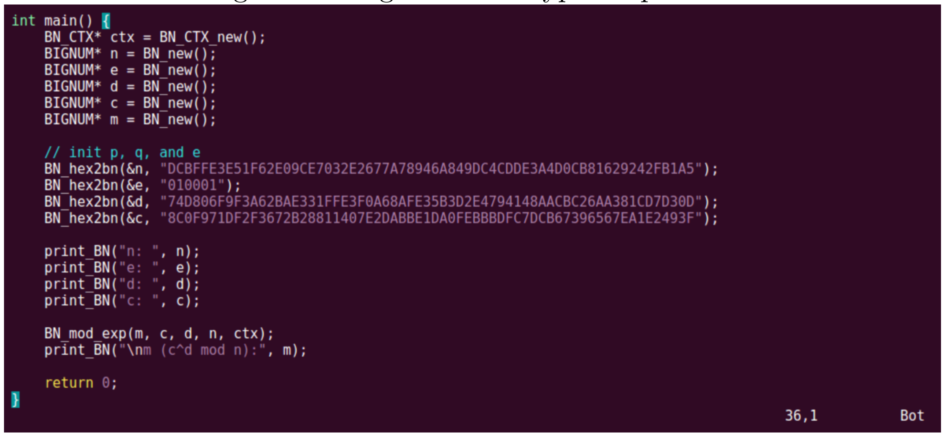
Task 3

Task 3 focuses only on decryption of a message. This is a scenario in which person A encrypts a message such that only person B can decrypt the message. The program shown in Figure 5 is written from the perspective of person B (since the private key is known). The message that person B decrypts is

$$C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F$$

The decryption is performed by computing $C^d \mod n$, where d and n are taken from Task 2.

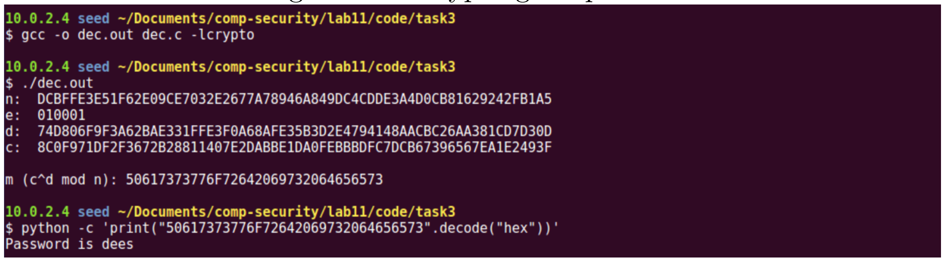
Figure 5: Program to decrypt a ciphertext



```
int main() {
    BN_CTX* ctx = BN_CTX_new();
    BIGNUM* n = BN_new();
    BIGNUM* e = BN_new();
    BIGNUM* d = BN_new();
    BIGNUM* c = BN_new();
    BIGNUM* m = BN_new();

    // init p, q, and e
    BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
    BN_hex2bn(&c, "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");

    print_BN("n: ", n);
    print_BN("e: ", e);
    print_BN("d: ", d);
    print_BN("c: ", c);

    BN_mod_exp(m, c, d, n, ctx);
    print_BN("\nm (c^d mod n):", m);

    return 0;
}
                                                                  36,1          Bot
```

The output of this process is shown in Figure 6. The message is "Password is dees."

Figure 6: Decrypting a ciphertext



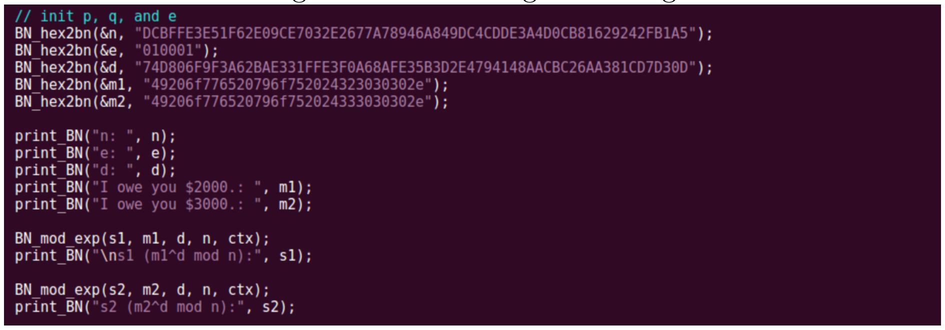
```
10.0.2.4 seed ~/Documents/comp-security/lab11/code/task3
$ gcc -o dec.out dec.c -lcrypto

10.0.2.4 seed ~/Documents/comp-security/lab11/code/task3
$ ./dec.out
n:  DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e:  010001
d:  74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
c:  8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F

m (c^d mod n): 50617373776F72642069732064656573

10.0.2.4 seed ~/Documents/comp-security/lab11/code/task3
$ python -c 'print("50617373776F72642069732064656573".decode("hex"))'
Password is dees
```

Task 4

Task 4 illuminates another use for public/private key cryptography: digital signatures. Since the order of exponentiation does not affect the final result (ie $(m^a \mod n)^b \mod n = (m^b \mod n)^a \mod n$), person B could encrypt a message using their private key and everyone else could use their public key to decrypt the message and know that the message came from person B.

Code to do this is shown in Figure 7. Note that there are two messages in this code, and they differ only by 1 bit! Will their signatures be similar?

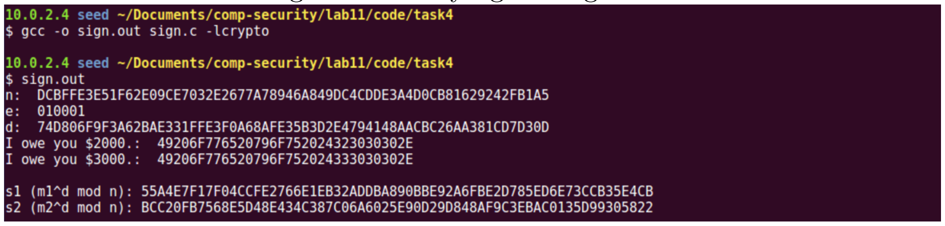
Figure 7: Code to sign a message



```
// init p, q, and e
BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
BN_hex2bn(&e, "010001");
BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
BN_hex2bn(&m1, "49206f776520796f752024323030302e");
BN_hex2bn(&m2, "49206f776520796f752024333030302e");

print_BN("n: ", n);
print_BN("e: ", e);
print_BN("d: ", d);
print_BN("I owe you $2000.: ", m1);
print_BN("I owe you $3000.: ", m2);

BN_mod_exp(s1, m1, d, n, ctx);
print_BN("\ns1 (m1^d mod n):", s1);

BN_mod_exp(s2, m2, d, n, ctx);
print_BN("s2 (m2^d mod n):", s2);
```

The signatures are radically different, as shown in Figure 8. This means that an attacker would have trouble changing a message and convincing people that the message was from a particular person (since they would not know the private key).

Figure 8: Verifying the signatures

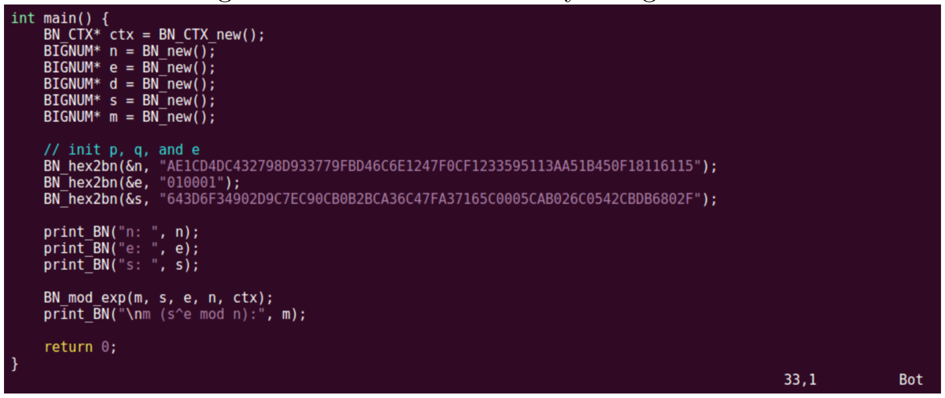


```
10.0.2.4 seed ~/Documents/comp-security/lab11/code/task4
$ gcc -o sign.out sign.c -lcrypto

10.0.2.4 seed ~/Documents/comp-security/lab11/code/task4
$ sign.out
n:  DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e:  010001
d:  74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
I owe you $2000.:  49206F776520796F752024323030302E
I owe you $3000.:  49206F776520796F752024333030302E

s1 (m1^d mod n): 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
s2 (m2^d mod n): BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135D99305822
```

Task 5

Task 5 focuses on testing the validity of messages by way of digital signatures. The person decrypts the message using the public key of the sender (computing $s^e \mod n$), and then interprets the message. The code to do this is shown in Figure 9.

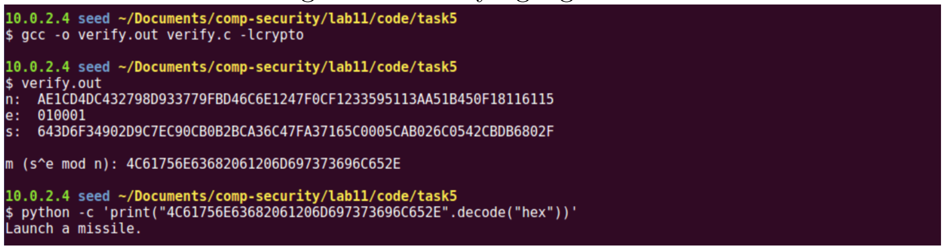
Figure 9: Code to test validity of signature 1



```
int main() {
    BN_CTX* ctx = BN_CTX_new();
    BIGNUM* n = BN_new();
    BIGNUM* e = BN_new();
    BIGNUM* d = BN_new();
    BIGNUM* s = BN_new();
    BIGNUM* m = BN_new();

    // init p, q, and e
    BN_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&s, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");

    print_BN("n: ", n);
    print_BN("e: ", e);
    print_BN("s: ", s);

    BN_mod_exp(m, s, e, n, ctx);
    print_BN("\nm (s^e mod n):", m);

    return 0;
}
                                                                  33,1          Bot
```

The output of this program shows that decrypted signature and then converts the hex back to ascii. Figure 10 shows that the decrypted message matches in the original message.

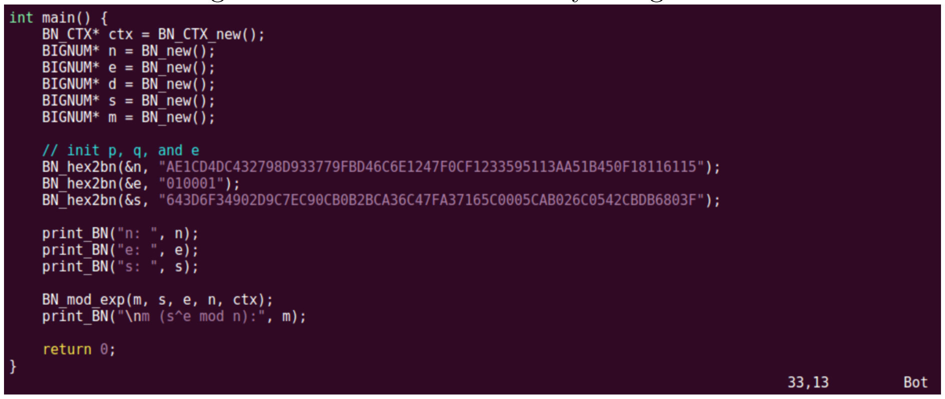
Figure 10: Verifying signature 1



```
10.0.2.4 seed ~/Documents/comp-security/lab11/code/task5
$ gcc -o verify.out verify.c -lcrypto

10.0.2.4 seed ~/Documents/comp-security/lab11/code/task5
$ verify.out
n:  AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
e:  010001
s:  643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F

m (s^e mod n): 4C61756E63682061206D697373696C652E

10.0.2.4 seed ~/Documents/comp-security/lab11/code/task5
$ python -c 'print("4C61756E63682061206D697373696C652E".decode("hex"))'
Launch a missile.
```

Figure 11 explores the possibility that the signature has been tampered with. Say that a small change was made to the signature, what will happen to the decrypted message? The updated code (ie changing the cipher text) is shown in Figure 11.

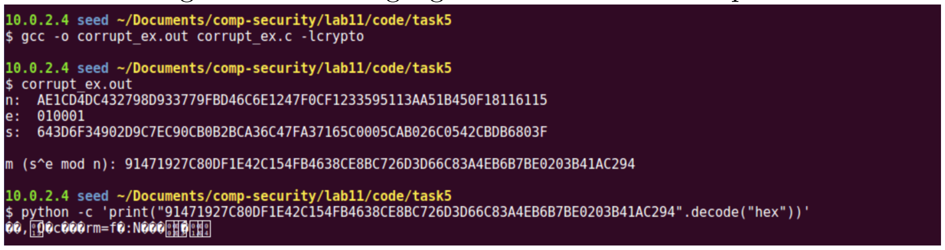
Figure 11: Code to test validity of signature 2



```
int main() {
    BN_CTX* ctx = BN_CTX_new();
    BIGNUM* n = BN_new();
    BIGNUM* e = BN_new();
    BIGNUM* d = BN_new();
    BIGNUM* s = BN_new();
    BIGNUM* m = BN_new();

    // init p, q, and e
    BN_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
    BN_hex2bn(&e, "010001");
    BN_hex2bn(&s, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");

    print_BN("n: ", n);
    print_BN("e: ", e);
    print_BN("s: ", s);

    BN_mod_exp(m, s, e, n, ctx);
    print_BN("\nm (s^e mod n):", m);

    return 0;
}
                                                                  33,13         Bot
```

The output of the code is shown in Figure 12. Here, it can be seen that the decrypted message is entirely gibberish when converted back to ascii. So small changes in the cipher text can cause large changes when converting back to plain text.

Figure 12: Showing signature 2 has been tampered



```
10.0.2.4 seed ~/Documents/comp-security/lab11/code/task5
$ gcc -o corrupt_ex.out corrupt_ex.c -lcrypto

10.0.2.4 seed ~/Documents/comp-security/lab11/code/task5
$ corrupt_ex.out
n:  AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
e:  010001
s:  643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F

m (s^e mod n): 91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294

10.0.2.4 seed ~/Documents/comp-security/lab11/code/task5
$ python -c 'print("91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294".decode("hex"))'
��,��c���rm=f�:N���
```

This lab explored some of the uses of the RSA encryption system. The math behind this encryption/decryption process is quite sound and is used regularly in our age of internet.