D0029E - Asymmetric Encryption (Lab 3)

Martin Askolin[*]

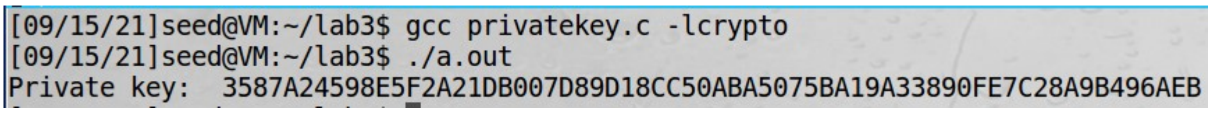
Luleå tekniska universitet
971 87 Luleå, Sverige

15 september 2021

[*]email: `marsak-8@student.ltu.se`

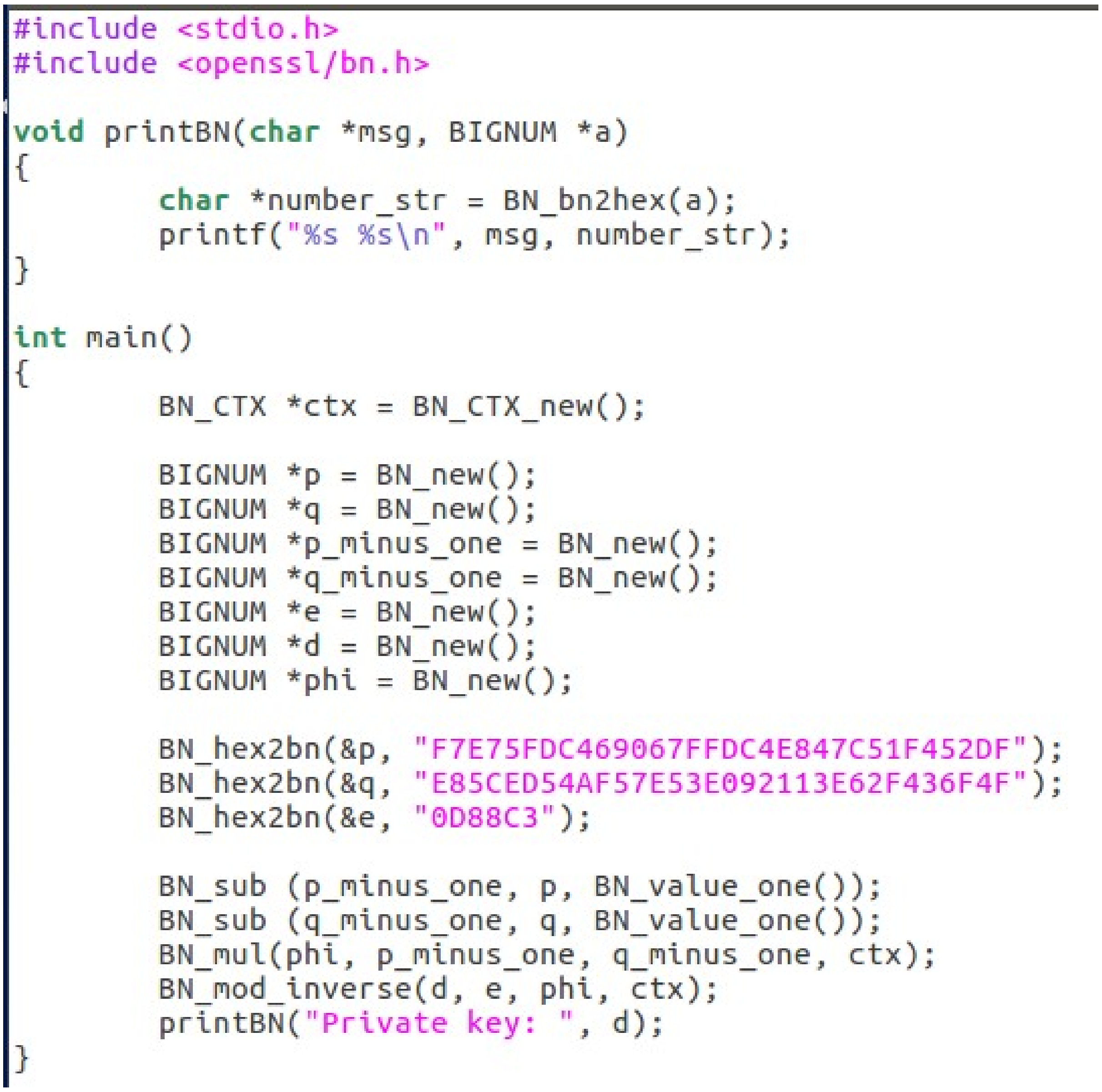
1 Deriving the private key

We get the private key from solving $e \cdot d \bmod \phi(n) = 1$.



```
[09/15/21]seed@VM:~/lab3$ gcc privatekey.c -lcrypto
[09/15/21]seed@VM:~/lab3$ ./a.out
Private key:  3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
```

Figur 1: Bash commands.

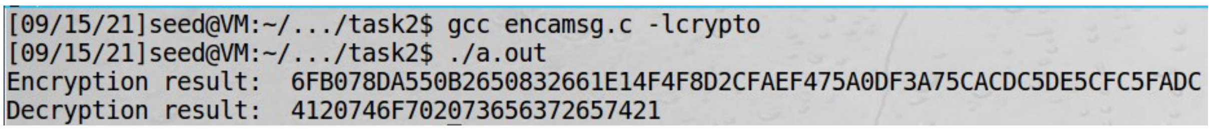


```c
#include <stdio.h>
#include <openssl/bn.h>

void printBN(char *msg, BIGNUM *a)
{
        char *number_str = BN_bn2hex(a);
        printf("%s %s\n", msg, number_str);
}

int main()
{
        BN_CTX *ctx = BN_CTX_new();

        BIGNUM *p = BN_new();
        BIGNUM *q = BN_new();
        BIGNUM *p_minus_one = BN_new();
        BIGNUM *q_minus_one = BN_new();
        BIGNUM *e = BN_new();
        BIGNUM *d = BN_new();
        BIGNUM *phi = BN_new();

        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(phi, p_minus_one, q_minus_one, ctx);
        BN_mod_inverse(d, e, phi, ctx);
        printBN("Private key: ", d);
}
```

Figur 2: Code for getting the private key.

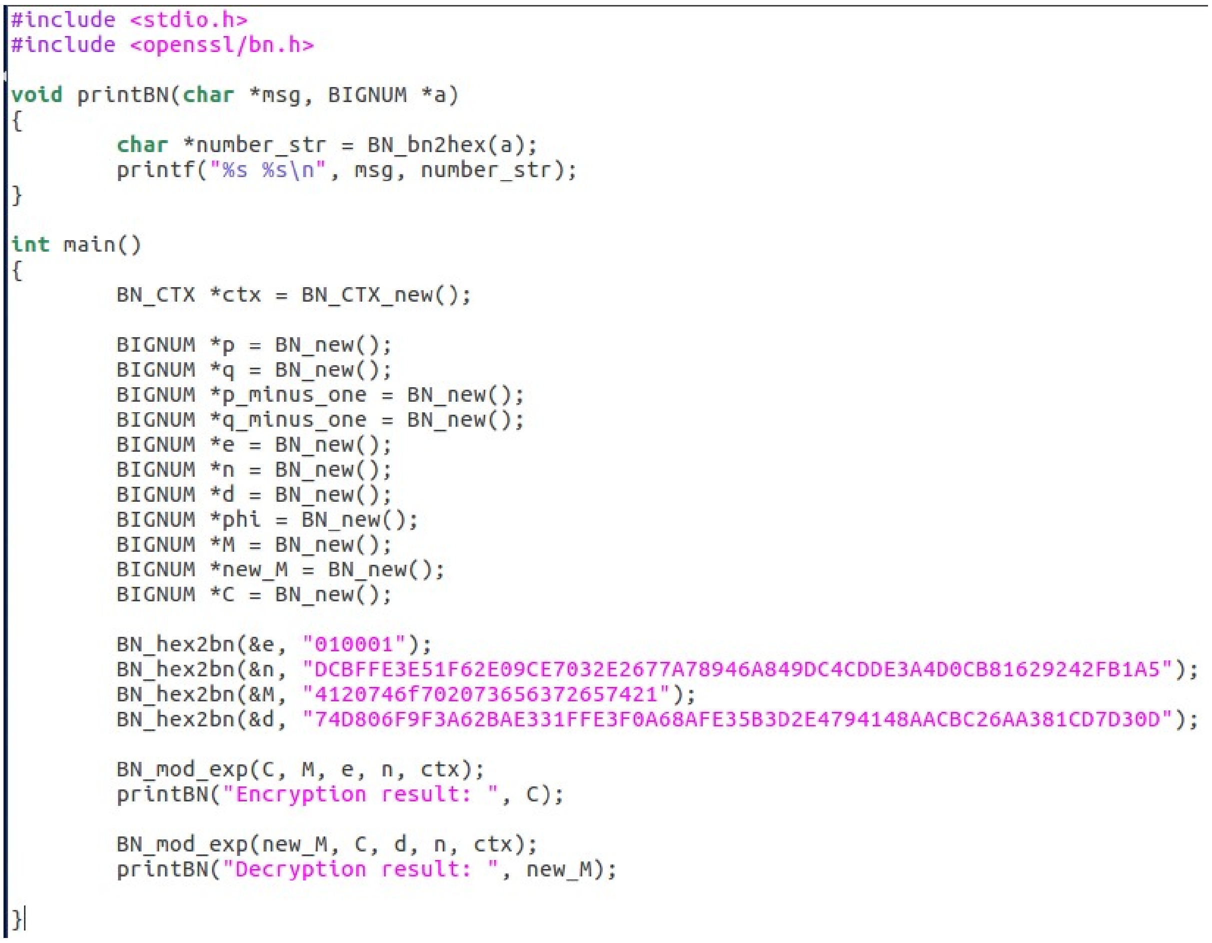
2 Encrypting a Message

After decrypting our encrypted message we get back 'A top secret!' in hex.



```
[09/15/21]seed@VM:~/.../task2$ gcc encamsg.c -lcrypto
[09/15/21]seed@VM:~/.../task2$ ./a.out
Encryption result:  6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
Decryption result:  4120746F702073656372657421
```

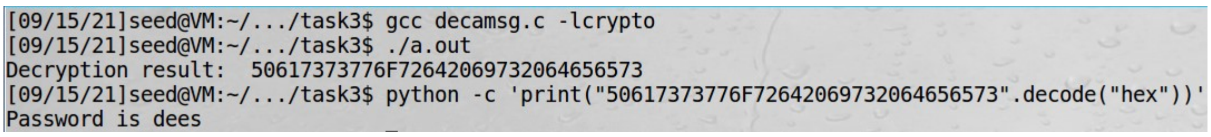
Figur 3: Bash commands



```c
#include <stdio.h>
#include <openssl/bn.h>

void printBN(char *msg, BIGNUM *a)
{
        char *number_str = BN_bn2hex(a);
        printf("%s %s\n", msg, number_str);
}

int main()
{
        BN_CTX *ctx = BN_CTX_new();

        BIGNUM *p = BN_new();
        BIGNUM *q = BN_new();
        BIGNUM *p_minus_one = BN_new();
        BIGNUM *q_minus_one = BN_new();
        BIGNUM *e = BN_new();
        BIGNUM *n = BN_new();
        BIGNUM *d = BN_new();
        BIGNUM *phi = BN_new();
        BIGNUM *M = BN_new();
        BIGNUM *new_M = BN_new();
        BIGNUM *C = BN_new();

        BN_hex2bn(&e, "010001");
        BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
        BN_hex2bn(&M, "4120746f702073656372657421");
        BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

        BN_mod_exp(C, M, e, n, ctx);
        printBN("Encryption result: ", C);

        BN_mod_exp(new_M, C, d, n, ctx);
        printBN("Decryption result: ", new_M);

}
```

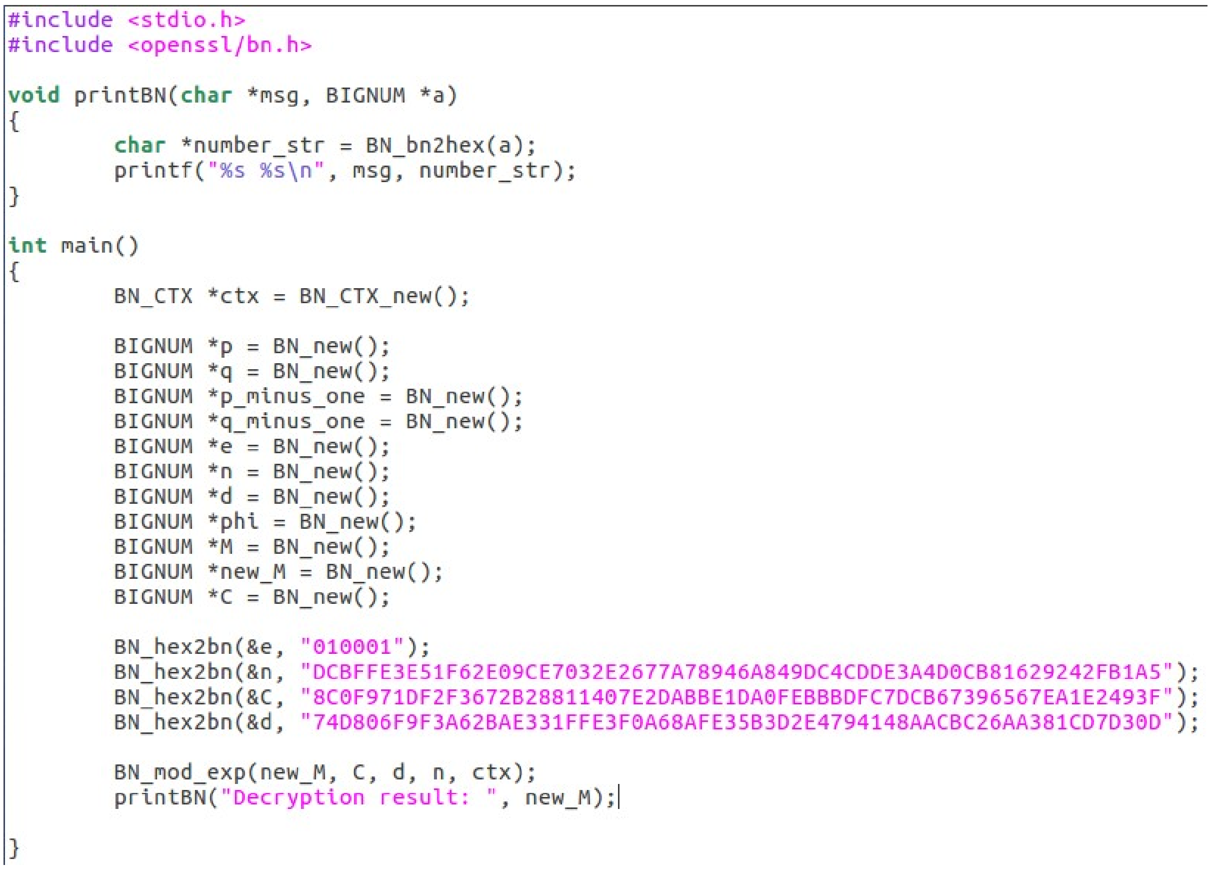
Figur 4: Code for encrypting the message

3 Decrypting a Message



```
[09/15/21]seed@VM:~/.../task3$ gcc decamsg.c -lcrypto
[09/15/21]seed@VM:~/.../task3$ ./a.out
Decryption result:  50617373776F72642069732064656573
[09/15/21]seed@VM:~/.../task3$ python -c 'print("50617373776F72642069732064656573".decode("hex"))'
Password is dees
```

Figur 5: Bash commands

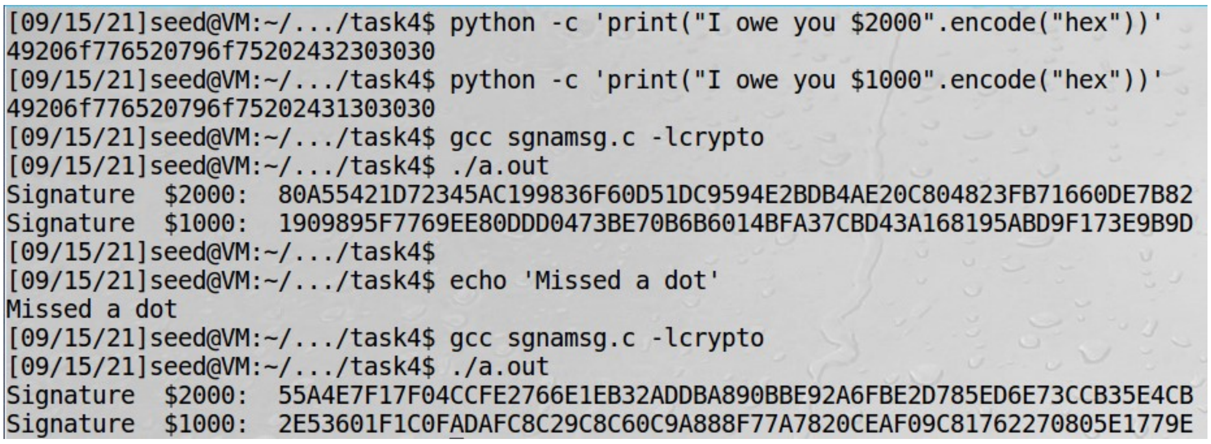


```c
#include <stdio.h>
#include <openssl/bn.h>

void printBN(char *msg, BIGNUM *a)
{
	char *number_str = BN_bn2hex(a);
	printf("%s %s\n", msg, number_str);
}

int main()
{
	BN_CTX *ctx = BN_CTX_new();

	BIGNUM *p = BN_new();
	BIGNUM *q = BN_new();
	BIGNUM *p_minus_one = BN_new();
	BIGNUM *q_minus_one = BN_new();
	BIGNUM *e = BN_new();
	BIGNUM *n = BN_new();
	BIGNUM *d = BN_new();
	BIGNUM *phi = BN_new();
	BIGNUM *M = BN_new();
	BIGNUM *new_M = BN_new();
	BIGNUM *C = BN_new();

	BN_hex2bn(&e, "010001");
	BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
	BN_hex2bn(&C, "8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");
	BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

	BN_mod_exp(new_M, C, d, n, ctx);
	printBN("Decryption result: ", new_M);

}
```

Figur 6: Code for decrypting the message

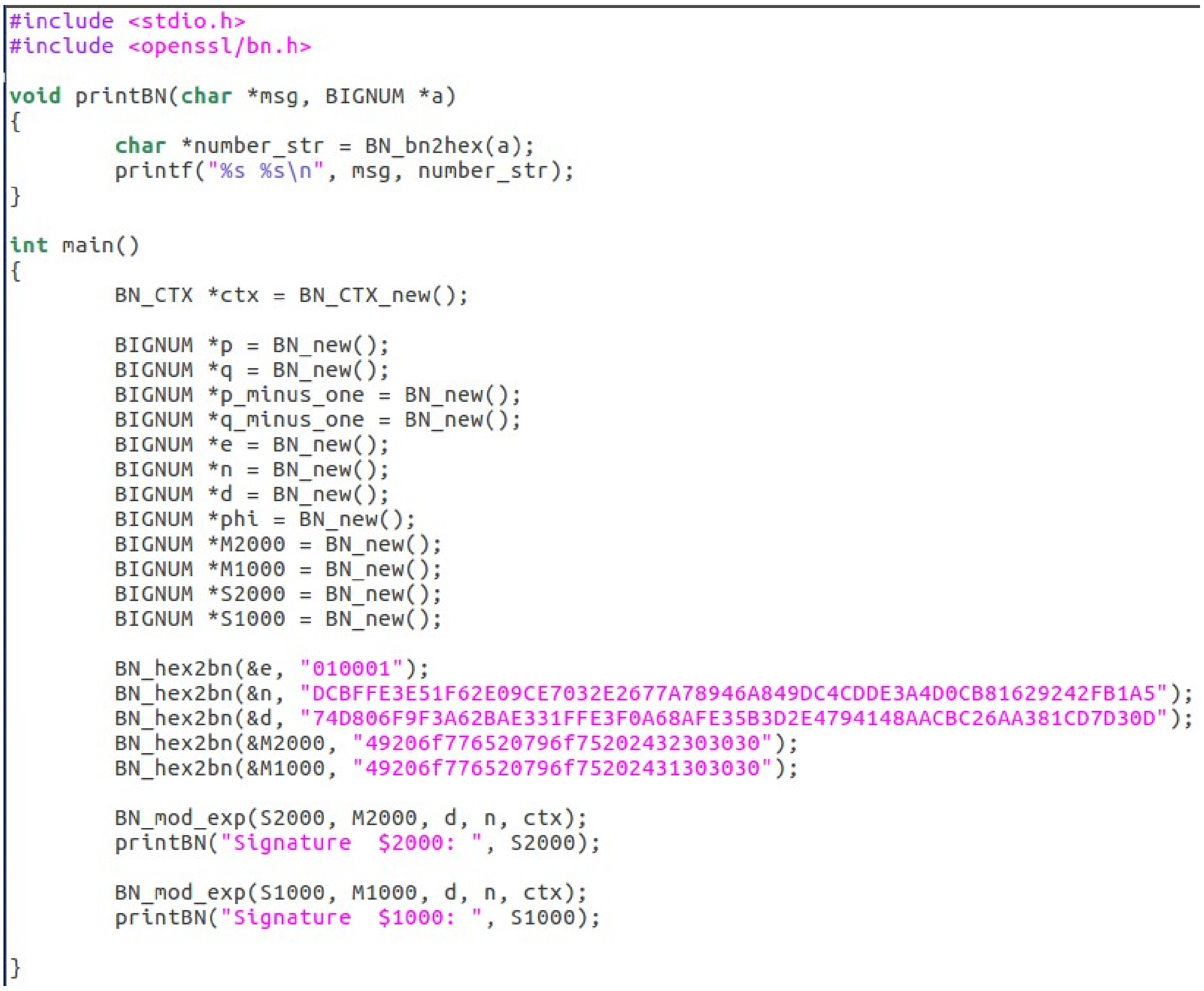
4 Signing a message

From Figure 7 it can be noted that by just changing one byte; changing it from 2000 to 1000, the cipher texts come out nothing alike. You would not be able to tell just how similar these two messages are thanks to $C = M^e \bmod n$.



```
[09/15/21]seed@VM:~/.../task4$ python -c 'print("I owe you $2000".encode("hex"))'
49206f776520796f75202432303030
[09/15/21]seed@VM:~/.../task4$ python -c 'print("I owe you $1000".encode("hex"))'
49206f776520796f75202431303030
[09/15/21]seed@VM:~/.../task4$ gcc sgnamsg.c -lcrypto
[09/15/21]seed@VM:~/.../task4$ ./a.out
Signature  $2000:  80A55421D72345AC199836F60D51DC9594E2BDB4AE20C804823FB71660DE7B82
Signature  $1000:  1909895F7769EE80DDD0473BE70B6B6014BFA37CBD43A168195ABD9F173E9B9D
[09/15/21]seed@VM:~/.../task4$
[09/15/21]seed@VM:~/.../task4$ echo 'Missed a dot'
Missed a dot
[09/15/21]seed@VM:~/.../task4$ gcc sgnamsg.c -lcrypto
[09/15/21]seed@VM:~/.../task4$ ./a.out
Signature  $2000:  55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73CCB35E4CB
Signature  $1000:  2E53601F1C0FADAFC8C29C8C60C9A888F77A7820CEAF09C81762270805E1779E
```

Figur 7: Bash commands used.

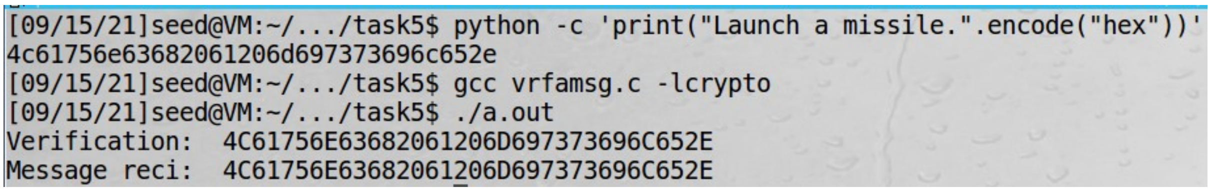


```c
#include <stdio.h>
#include <openssl/bn.h>

void printBN(char *msg, BIGNUM *a)
{
        char *number_str = BN_bn2hex(a);
        printf("%s %s\n", msg, number_str);
}

int main()
{
        BN_CTX *ctx = BN_CTX_new();

        BIGNUM *p = BN_new();
        BIGNUM *q = BN_new();
        BIGNUM *p_minus_one = BN_new();
        BIGNUM *q_minus_one = BN_new();
        BIGNUM *e = BN_new();
        BIGNUM *n = BN_new();
        BIGNUM *d = BN_new();
        BIGNUM *phi = BN_new();
        BIGNUM *M2000 = BN_new();
        BIGNUM *M1000 = BN_new();
        BIGNUM *S2000 = BN_new();
        BIGNUM *S1000 = BN_new();

        BN_hex2bn(&e, "010001");
        BN_hex2bn(&n, "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");
        BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
        BN_hex2bn(&M2000, "49206f776520796f75202432303030");
        BN_hex2bn(&M1000, "49206f776520796f75202431303030");

        BN_mod_exp(S2000, M2000, d, n, ctx);
        printBN("Signature  $2000: ", S2000);

        BN_mod_exp(S1000, M1000, d, n, ctx);
        printBN("Signature  $1000: ", S1000);

}
```

Figur 8: Code for signing a message.

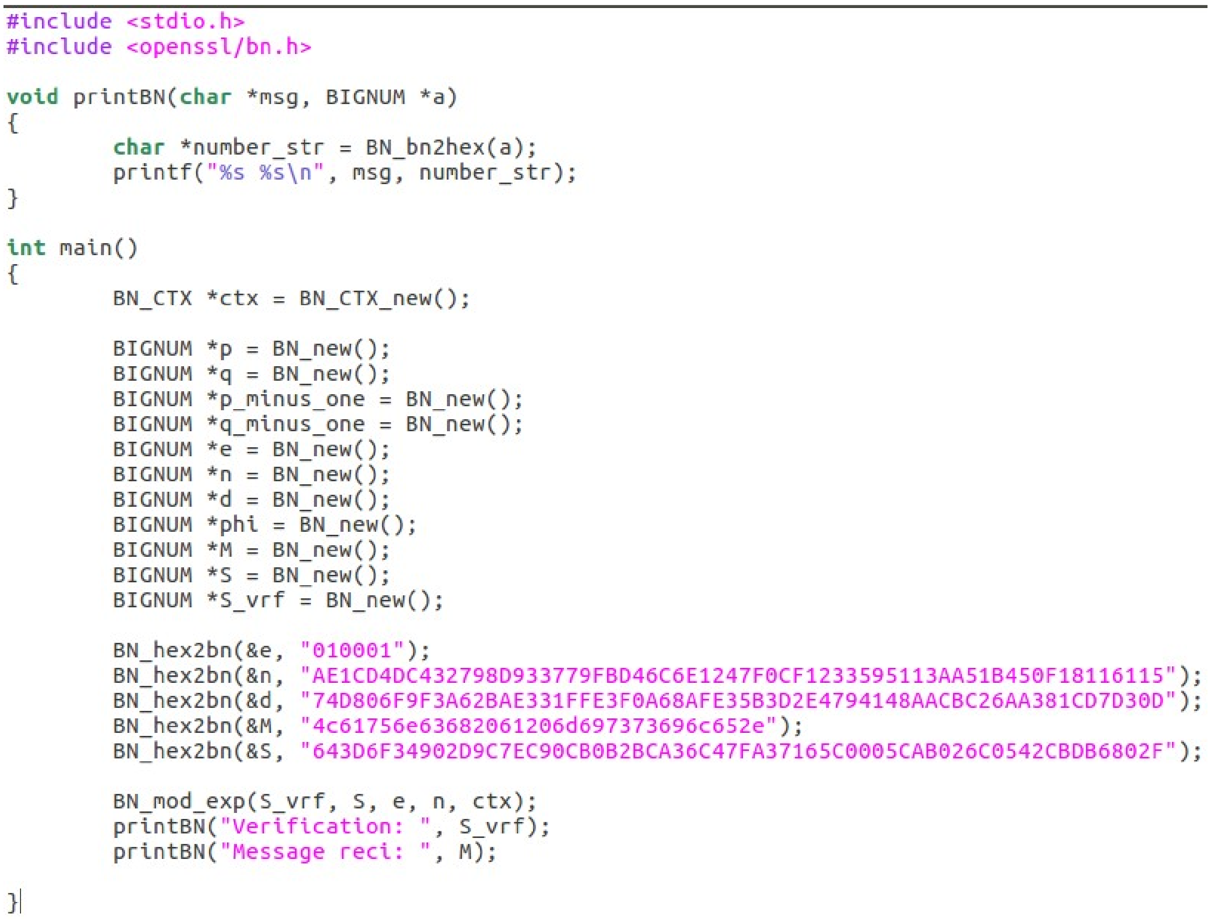
5 Verifying a Signature

The message is verified and I will proceed to launch the missile. Since Alice is the only one in possession of the private key and the message has not been tampered with I am confident that launching this missile is precisely what Alice wants!



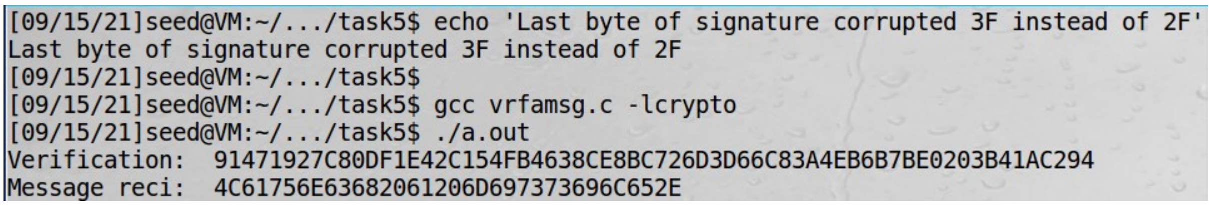
```
[09/15/21]seed@VM:~/.../task5$ python -c 'print("Launch a missile.".encode("hex"))'
4c61756e63682061206d697373696c652e
[09/15/21]seed@VM:~/.../task5$ gcc vrfamsg.c -lcrypto
[09/15/21]seed@VM:~/.../task5$ ./a.out
Verification:  4C61756E63682061206D697373696C652E
Message reci:  4C61756E63682061206D697373696C652E
```

Figur 9: Bash commands used.



```c
#include <stdio.h>
#include <openssl/bn.h>

void printBN(char *msg, BIGNUM *a)
{
        char *number_str = BN_bn2hex(a);
        printf("%s %s\n", msg, number_str);
}

int main()
{
        BN_CTX *ctx = BN_CTX_new();

        BIGNUM *p = BN_new();
        BIGNUM *q = BN_new();
        BIGNUM *p_minus_one = BN_new();
        BIGNUM *q_minus_one = BN_new();
        BIGNUM *e = BN_new();
        BIGNUM *n = BN_new();
        BIGNUM *d = BN_new();
        BIGNUM *phi = BN_new();
        BIGNUM *M = BN_new();
        BIGNUM *S = BN_new();
        BIGNUM *S_vrf = BN_new();

        BN_hex2bn(&e, "010001");
        BN_hex2bn(&n, "AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");
        BN_hex2bn(&d, "74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");
        BN_hex2bn(&M, "4c61756e63682061206d697373696c652e");
        BN_hex2bn(&S, "643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");

        BN_mod_exp(S_vrf, S, e, n, ctx);
        printBN("Verification: ", S_vrf);
        printBN("Message reci: ", M);

}
```

Figur 10: Code for verifying a message.



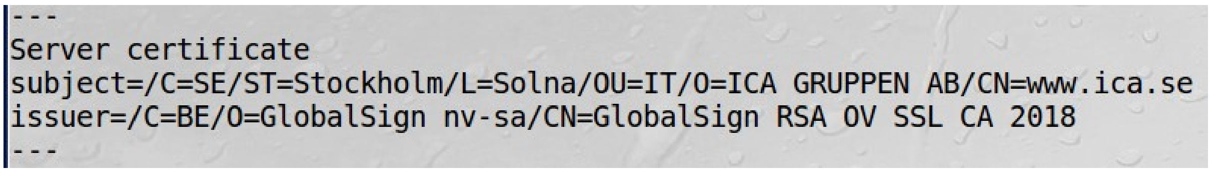
```
[09/15/21]seed@VM:~/.../task5$ echo 'Last byte of signature corrupted 3F instead of 2F'
Last byte of signature corrupted 3F instead of 2F
[09/15/21]seed@VM:~/.../task5$
[09/15/21]seed@VM:~/.../task5$ gcc vrfamsg.c -lcrypto
[09/15/21]seed@VM:~/.../task5$ ./a.out
Verification:  91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
Message reci:  4C61756E6368206120 6D697373696C652E
```

Figur 11: Corrupted / tampered message.

Since the signature was corrupted it did not map correctly to the original message. Since $C = M^e \bmod n$ and n has a length of 32 bytes it is very plausible that this verification is longer or shorter (in this case longer) than the 17 byte message we were expecting.

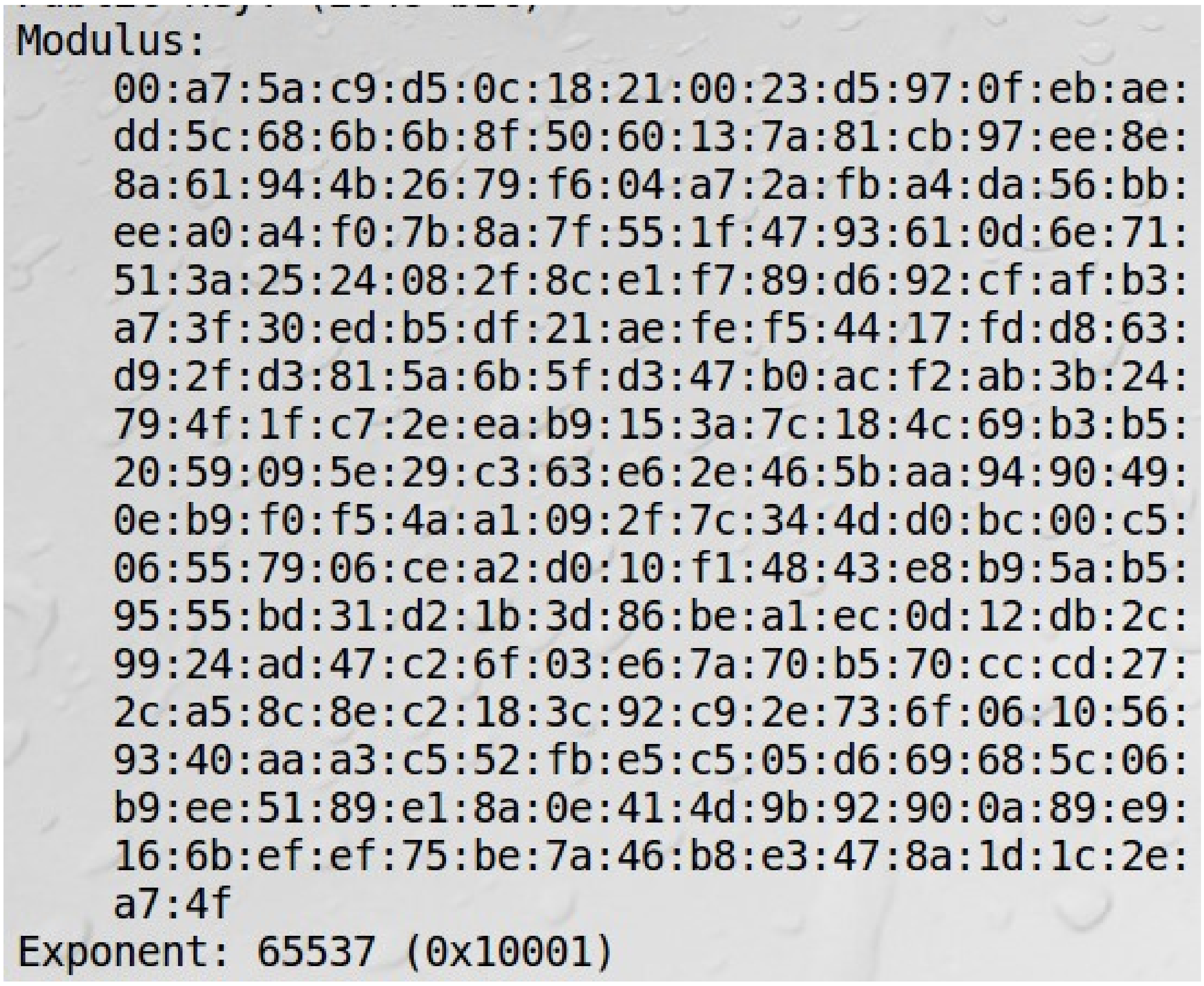
6 Manually verifying an X.509 certificate

After receiving the public key, certificate and signature we use the code in Figure 15 to verify the validity of the signature.



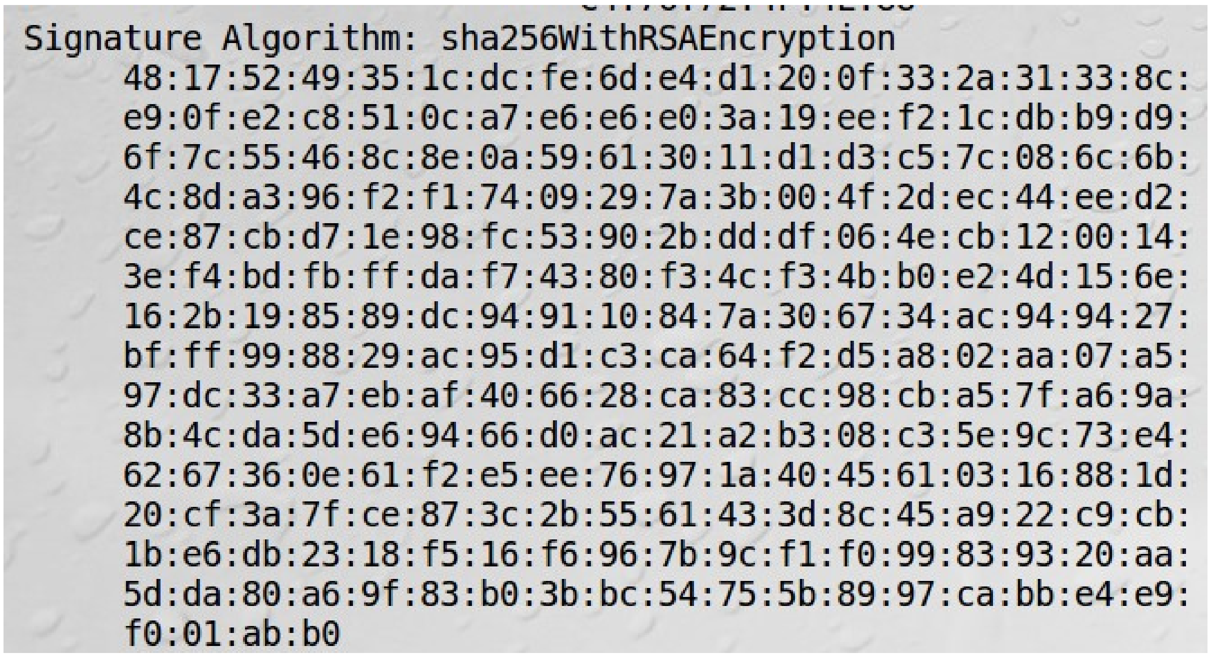
```
---
Server certificate
subject=/C=SE/ST=Stockholm/L=Solna/OU=IT/O=ICA GRUPPEN AB/CN=www.ica.se
issuer=/C=BE/O=GlobalSign nv-sa/CN=GlobalSign RSA OV SSL CA 2018
---
```

Figur 12: I choose to verify 'www.ica.se' (Popular swedish grocery store.)



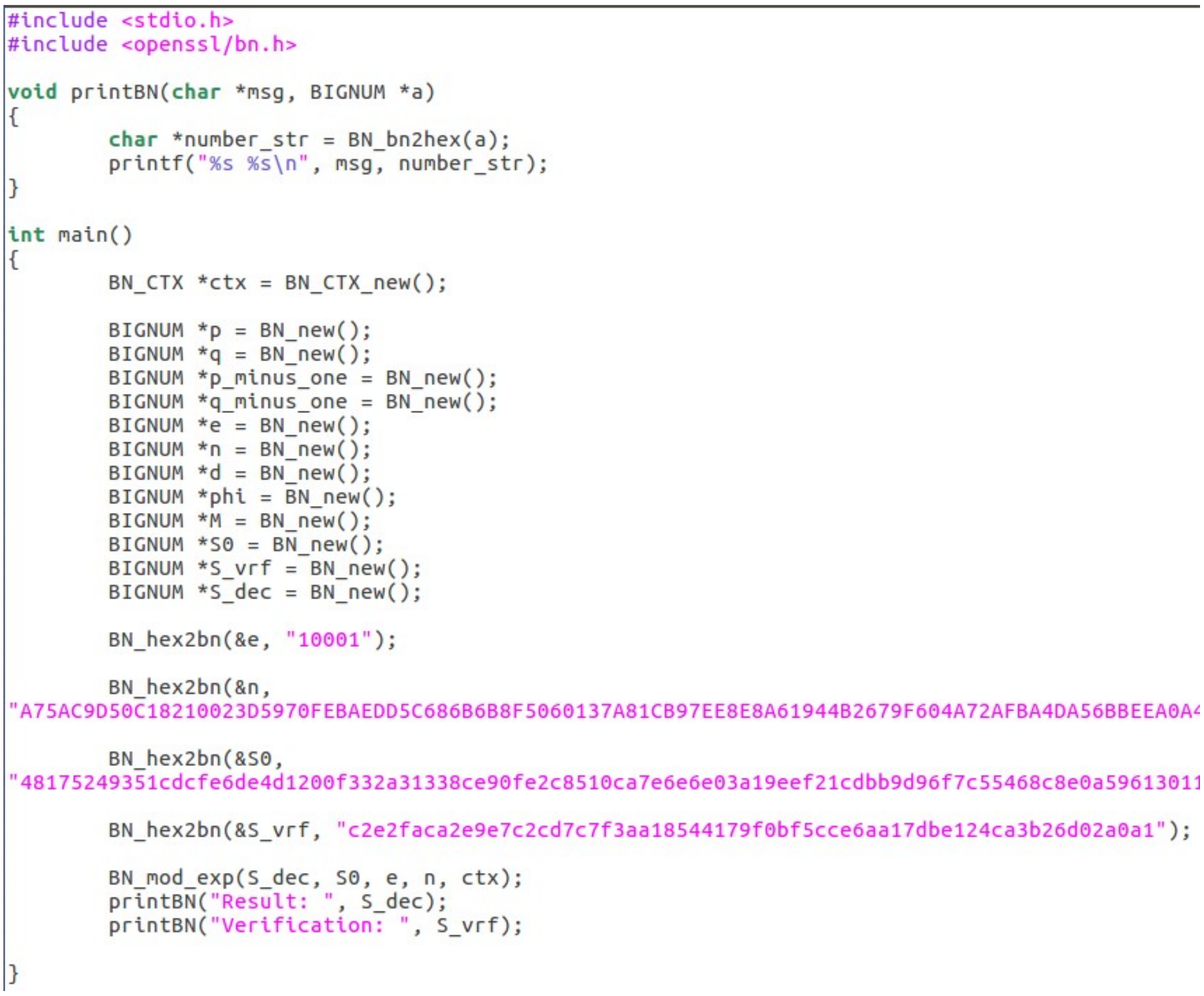
```
Modulus:
    00:a7:5a:c9:d5:0c:18:21:00:23:d5:97:0f:eb:ae:
    dd:5c:68:6b:6b:8f:50:60:13:7a:81:cb:97:ee:8e:
    8a:61:94:4b:26:79:f6:04:a7:2a:fb:a4:da:56:bb:
    ee:a0:a4:f0:7b:8a:7f:55:1f:47:93:61:0d:6e:71:
    51:3a:25:24:08:2f:8c:e1:f7:89:d6:92:cf:af:b3:
    a7:3f:30:ed:b5:df:21:ae:fe:f5:44:17:fd:d8:63:
    d9:2f:d3:81:5a:6b:5f:d3:47:b0:ac:f2:ab:3b:24:
    79:4f:1f:c7:2e:ea:b9:15:3a:7c:18:4c:69:b3:b5:
    20:59:09:5e:29:c3:63:e6:2e:46:5b:aa:94:90:49:
    0e:b9:f0:f5:4a:a1:09:2f:7c:34:4d:d0:bc:00:c5:
    06:55:79:06:ce:a2:d0:10:f1:48:43:e8:b9:5a:b5:
    95:55:bd:31:d2:1b:3d:86:be:a1:ec:0d:12:db:2c:
    99:24:ad:47:c2:6f:03:e6:7a:70:b5:70:cc:cd:27:
    2c:a5:8c:8e:c2:18:3c:92:c9:2e:73:6f:06:10:56:
    93:40:aa:a3:c5:52:fb:e5:c5:05:d6:69:68:5c:06:
    b9:ee:51:89:e1:8a:0e:41:4d:9b:92:90:0a:89:e9:
    16:6b:ef:ef:75:be:7a:46:b8:e3:47:8a:1d:1c:2e:
    a7:4f
Exponent: 65537 (0x10001)
```

Figur 13: Modulus, n and exponent, e.



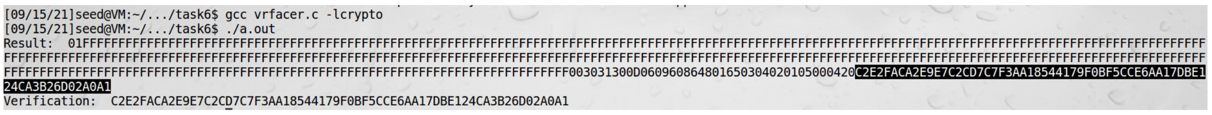
```
Signature Algorithm: sha256WithRSAEncryption
     48:17:52:49:35:1c:dc:fe:6d:e4:d1:20:0f:33:2a:31:33:8c:
     e9:0f:e2:c8:51:0c:a7:e6:e6:e0:3a:19:ee:f2:1c:db:b9:d9:
     6f:7c:55:46:8c:8e:0a:59:61:30:11:d1:d3:c5:7c:08:6c:6b:
     4c:8d:a3:96:f2:f1:74:09:29:7a:3b:00:4f:2d:ec:44:ee:d2:
     ce:87:cb:d7:1e:98:fc:53:90:2b:dd:df:06:4e:cb:12:00:14:
     3e:f4:bd:fb:ff:da:f7:43:80:f3:4c:f3:4b:b0:e2:4d:15:6e:
     16:2b:19:85:89:dc:94:91:10:84:7a:30:67:34:ac:94:94:27:
     bf:ff:99:88:29:ac:95:d1:c3:ca:64:f2:d5:a8:02:aa:07:a5:
     97:dc:33:a7:eb:af:40:66:28:ca:83:cc:98:cb:a5:7f:a6:9a:
     8b:4c:da:5d:e6:94:66:d0:ac:21:a2:b3:08:c3:5e:9c:73:e4:
     62:67:36:0e:61:f2:e5:ee:76:97:1a:40:45:61:03:16:88:1d:
     20:cf:3a:7f:ce:87:3c:2b:55:61:43:3d:8c:45:a9:22:c9:cb:
     1b:e6:db:23:18:f5:16:f6:96:7b:9c:f1:f0:99:83:93:20:aa:
     5d:da:80:a6:9f:83:b0:3b:bc:54:75:5b:89:97:ca:bb:e4:e9:
     f0:01:ab:b0
```

Figur 14: SHA hash function with RSA encryption + Signature.



```c
#include <stdio.h>
#include <openssl/bn.h>

void printBN(char *msg, BIGNUM *a)
{
        char *number_str = BN_bn2hex(a);
        printf("%s %s\n", msg, number_str);
}

int main()
{
        BN_CTX *ctx = BN_CTX_new();

        BIGNUM *p = BN_new();
        BIGNUM *q = BN_new();
        BIGNUM *p_minus_one = BN_new();
        BIGNUM *q_minus_one = BN_new();
        BIGNUM *e = BN_new();
        BIGNUM *n = BN_new();
        BIGNUM *d = BN_new();
        BIGNUM *phi = BN_new();
        BIGNUM *M = BN_new();
        BIGNUM *S0 = BN_new();
        BIGNUM *S_vrf = BN_new();
        BIGNUM *S_dec = BN_new();

        BN_hex2bn(&e, "10001");

        BN_hex2bn(&n,
"A75AC9D50C18210023D5970FEBAEDD5C686B6B8F5060137A81CB97EE8E8A61944B2679F604A72AFBA4DA56BBEEA0A

        BN_hex2bn(&S0,
"48175249351cdcfe6de4d1200f332a31338ce90fe2c8510ca7e6e6e03a19eef21cdbb9d96f7c55468c8e0a5961301

        BN_hex2bn(&S_vrf, "c2e2faca2e9e7c2cd7c7f3aa18544179f0bf5cce6aa17dbe124ca3b26d02a0a1");

        BN_mod_exp(S_dec, S0, e, n, ctx);
        printBN("Result: ", S_dec);
        printBN("Verification: ", S_vrf);

}
```

Figur 15: Code for verifying the signature.



```
[09/15/21]seed@VM:~/.../task6$ gcc vrfacer.c -lcrypto
[09/15/21]seed@VM:~/.../task6$ ./a.out
Result:  01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF003031300D060960864801650304020105000420C2E2FACA2E9E7C2CD7C7F3AA18544179F0BF5CCE6AA17DBE1
24CA3B26D02A0A1
Verification:  C2E2FACA2E9E7C2CD7C7F3AA18544179F0BF5CCE6AA17DBE124CA3B26D02A0A1
```

Figur 16: The signature is valid!