Eduard Klimenko
Anthony Trang
Ryan Moe
Public Key Lab

Task 1: Deriving the Private Key

Here we can see the specified p, q, and e, followed by the d they are meant to generate using the extended euclidean algorithm.

```
[04/30/19]seed@VM:~/.../hw3$ gcc bignum2.c -lcrypto
[04/30/19]seed@VM:~/.../hw3$ ./a.out
p = F7E75FDC469067FFDC4E847C51F452DF
q = E85CED54AF57E53E092113E62F436F4F
n = E103ABD94892E3E74AFD724BF28E78366D9676BCCC70118BD0AA1968DBB143D1
e = 0D88C3
phi = E103ABD94892E3E74AFD724BF28E78348D52298BD687C44DEB3A81065A7981A4
d = 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
Public Key: e =0D88C3, n =E103ABD94892E3E74AFD724BF28E78366D9676BCCC70118BD0AA1968DBB143D1
Private Key: d = 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C28A9B496AEB
```

Task 2: Encrypting a Message

Here the plaintext m = “4120746f702073656372657421” is encrypted with the supplied n and e values, then successfully decrypted with d to produce an identical m.

```
[05/01/19]seed@VM:~/.../hw3$ ./a.out
Public Key: e = 010001, n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
Private Key: d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D

m = 4120746F702073656372657421
c = 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC
m decrypted = 4120746F702073656372657421
```

Task 3: Decrypting a Message

The same d is used to decrypt the supplied ciphertext. When converted back to ASCII, it reads “Password is dees”.

```
[05/01/19]seed@VM:~/.../hw3$ ./a.out
Public Key: e = 010001, n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
Private Key: d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D

c = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
m decrypted = 50617373776F72642069732064656573
[05/01/19]seed@VM:~/.../hw3$ echo 50617373776F72642069732064656573 | xxd -r -p
Password is dees[05/01/19]seed@VM:~/.../hw3$
```

Task 4: Signing a Message

Here we generate a signature by applying the private key to two plaintexts: “I owe you $2000.” and “I owe you $3000.”. As we can see, changing only one character entirely changes the signature.

```
[05/01/19]seed@VM:~/.../hw3$ ./a.out
Public Key: e = 010001, n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
Private Key: d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D

m = 49206F776520796F752024323030302E // I owe you $2000.
c = 3A759CBF53901AC41373EEC603955A8E6AF8D3BCD5E9F6DD62C873CBB675051E
m decrypted = 49206F776520796F752024323030302E
[05/01/19]seed@VM:~/.../hw3$ gcc bignum2.c -lcrypto
[05/01/19]seed@VM:~/.../hw3$ ./a.out
Public Key: e = 010001, n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
Private Key: d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D

m = 49206F776520796F752024333030302E // I owe you $3000.
c = D06908047527906C724937169FA68CE0AC442FEB99D1880438D331A88F44B074
m decrypted = 49206F776520796F752024333030302E
```

Task 5: Verifying a Signature

Here we verify the signature s of a message. By applying the public key, we accurately create the message “Launch a missile.”. However, by changing only a single hex character, the result after verifying is completely unreadable.

```
[05/01/19]seed@VM:~/.../hw3$ gcc bignum2.c -lcrypto
[05/01/19]seed@VM:~/.../hw3$ ./a.out
s = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
verify = 4C61756E63682061206D697373696C652E
[05/01/19]seed@VM:~/.../hw3$ echo 4C61756E63682061206D697373696C652E | xxd -r -p
Launch a missile.[05/01/19]seed@VM:~/.../hw3$
[05/01/19]seed@VM:~/.../hw3$ gcc bignum2.c -lcrypto
[05/01/19]seed@VM:~/.../hw3$ ./a.out
s = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F
verify = 91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294
[05/01/19]seed@VM:~/.../hw3$ echo 91471927C80DF1E42C154FB4638CE8BC726D3D66C83A4EB6B7BE0203B41AC294 | xxd
-r -p
��,�D�c���rm=f�:N�����[05/01/19]seed@VM:~/.../hw3$
```

Task 6: Manually Verifying an X.509 Certificate
Step 1:
Here are the certificates we downloaded from a real web server, the certificate snippet on the left is the web server's certificate we downloaded from. The one on the right is the intermediate CA.


```
-----BEGIN CERTIFICATE-----
MIIHQjCCBiqgAwIBAgIQCgYwQn9bvO1pVzllk7ZFHzANBgkqhkiG9w0BAQsFADB1
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMTQwMgYDVQQDEytEaWdpQ2VydCBTSEEyIEV4dGVuZGVk
IFZhbGlkYXRpb24gU2VydmVyIENBMB4XDTE4MDUwODAwMDAwMFoXDTIwMDYwMzEy
MDAwMFowgccxHTAbBgNVBA8MFFByaXZhdGUgT3JnYW5pemF0aW9uMRMwEQYLKwYB
BAGCNzwCAQMTAlVTMRkwFwYLKwYBBAGCNzwCAQITCERlbGF3YXJlMRAwDgYDVQQF
Ewc1MTU3NTUwMQswCQYDVQQGEwJVUzETMBEGA1UECBMKQ2FsaWZvcm5pYTEWMBQG
A1UEBxMNU2FuIEZyYW5jaXNjbzEVMBMGA1UEChMMR2l0SHViLCBJbmMuMRMwEQYD
VQQDEwpnaXRodWIuY29tMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA
xjyq8jyXDDrBTyitcnB90865tWBzpHSbindG/XqYQkzFMBlXmqkzC+FdTRBYyneZ
w5Pz+XWQvL+74JW6LsWNc2EF0xCEqLOJuC9zjPAqbr7uroNLghGxYf13YdqbG5oj
/4x+ogEG3dF/U5YIwVr658DKyESMV6eoYV9mDVfTuJastkqcwero+5ZAKfYVMLUE
sMwFtoTDJFmVf6JlkOWwsxp1WcQ/MRQK1cyqOoUFUgYylgdh3yeCDPeF22Ax8AlQ
xbcaI+GwfQL1FB7Jy+h+KjME9lE/UpgV6Qt2R1xNSmvFCBWu+NFX6epwFP/JRbkM
fLz0beYFUvmMgLtwVpEPSwIDAQABo4IDeTCCA3UwHwYDVR0jBBgwFoAUPdNQpdag
re7zSmAKZdMh1Pj41g8wHQYDVR0OBBYEFMnCU2FmnV+rJfQmzQ84mqhJ6kipMCUG
A1UdEQQeMByCCmdpdGh1Yi5jb22CDnd3dy5naXRodWIuY29tMA4GA1UdDwEB/wQE
AwIFoDAdBgNVHSUEFjAUBggrBgEFBQcDAQYIKwYBBQUHAwIwdQYDVR0fBG4wbDA0
oDKgMIYuaHR0cDovL2NybDMuZGlnaWNlcnQuY29tL3NoYTItZXYtc2VydmVyLWcy
LmNybDA0oDKgMIYuaHR0cDovL2NybDQuZGlnaWNlcnQuY29tL3NoYTItZXYtc2Vy
dmVyLWcyLmNybDBLBgNVHSAERDBCMDcGCWCGSAGG/WwCATAqMCgGCCsGAQUFBwIB
FhxodHRwczovL3d3dy5kaWdpY2VydC5jb20vQ1BTMAcGBWeBDAEBMIGIBggrBgEF
BQcBAQR8MHowJAYIKwYBBQUHMAGGGGh0dHA6Ly9vY3NwLmRpZ2ljZXJ0LmNvbTBS
BggrBgEFBQcwAoZGaHR0cDovL2NhY2VydHMuZGlnaWNlcnQuY29tL0RpZ2lDZXJ0
U0hBMkV4dGVuZGVkVmFsaWRhdGlvblNlcnZlckNBLmNydDAMBgNVHRMBAf8EAjAA
MIIBfgYKKwYBBAHWeQIEAgSCAW4EggFqAWgAdgCkuQmQtBhYFIe7E6LMZ3AKPDWY
BPkb37jjd80OyA3cEAAAAWNBYm0KAAAEAwBHMEUCIQDRZp38cTWsWH2GdBpe/uPT
Wnsu/m4BEC2+dIcvSykZYgIgCP5gGv6yzaazxBK2NwGdmmyuEFNSg2pARbMJlUFg
U5UAdgBWFAaaL9fC7NP14b1Esj7HRna5vJkRXMDvlJhV1onQ3QAAAWNBYm0tAAAE
AwBHMEUCIQCi7omUvYLm0b2LobtEeRAYnlIo7n6JxbYdrtYdmPUWJQIgVgw1AZ51
vK9ENinBg22FPxb82TvNDO05T17hxXRC2IYAdgC72d+8H4pxtZOUI5eqkntHOFeV
CqtS6BqQlmQ2jh7RhQAAAWNBYm3fAAAEAwBHMEUCIQChzdTKUU2N+XcqcK0OJYrN
8EYynloVxho4yPk6Dq3EPgIgdNH5u8rC3UcslQV4B9o0a0w204omDREGKTVuEpxG
eOQwDQYJKoZIhvcNAQELBQADggEBAHAPWpanWOW/ip2oJ5grAH8mqQfaunuCVE+v
ac+88lkDK/LVdFgl2B6kIHZiYClzKtfczG93hWvKbST4NRNHP9LiaQqdNC17e5vN
HnXVUGw+yxyjMLGqkgepOnZ2Rb14kcTOGp4i5AuJuuaMwXmCo7jUwPwfLe1NUlVB
Kqg6LK0Hcq4K0sZnxE8HFxiZ92WpV2AVWjRMEc/2z2shNoDvxvFUYyY1Oe67xINk
myQKc+ygSBZzyLnXSFVWmHr3u5dcaaQGGAR42v6Ydr4iL38Hd4dOiBma+FXsXBIq
WUjbST4VXmdaol7uzFMojA4zkxQDZAvF5XgJlAFadfySna/teik=
-----END CERTIFICATE-----
```

```
-----BEGIN CERTIFICATE-----
MIIEtjCCA56gAwIBAgIQDHmpRLCMEZUgkmFf4msdgzANBgkqhkiG9w0BAQsFADBs
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
d3cuZGlnaWNlcnQuY29tMSswKQYDVQQDEyJEaWdpQ2VydCBIaWdoIEFzc3VyYW5j
ZSBFViBSb290IENBMB4XDTEzMTAyMjEyMDAwMFoXDTI4MTAyMjEyMDAwMFowdTEL
MAkGA1UEBhMCVVMxFTATBgNVBAoTDERpZ2lDZXJ0IEluYzEZMBcGA1UECxMQd3d3
LmRpZ2ljZXJ0LmNvbTE0MDIGA1UEAxMrRGlnaUNlcnQgU0hBMiBFeHRlbmRlZCBW
YWxpZGF0aW9uIFNlcnZlciBDQTCCASIwDQYJKoZIhvcNAQEBBQADggEPADCCAQoC
ggEBANdTpARR+JmmFkhLZyeqk0nQOe0MsLAAh/FnKIaFjI5j2ryxQDji0/XspQUY
uD0+xZkXMuwYjPrxDKZkIYXLBxA0sFKIKx9om9KxjxKws9LniB8f7zh3VFNfgHk/
LhqqqB5LKw2rt2O5Nbd9FLxZS99RStKh4gzikIKHaq7q12TWmFXo/a8aUGxUvBHy
/Urynbt/DvTVvo4WiRJV2MBxNO723C3sxIclho3YIeSwTQyJ3DkmF93215SF2AQh
cJ1vb/9cuhnhRctWVyh+HA1BV6q3uCe7seT6Ku8hI3UarS2bhjWMnHe1c63YlC3k
8wyd7sFOYn4XwHGeLN7x+RAoGTMCAwEAAaOCAUkwggFFMBIGA1UdEwEB/wQIMAYB
Af8CAQAwDgYDVR0PAQH/BAQDAgGGMB0GA1UdJQQWMBQGCCsGAQUFBwMBBggrBgEF
BQcDAjA0BggrBgEFBQcBAQQoMCYwJAYIKwYBBQUHMAGGGGh0dHA6Ly9vY3NwLmRp
Z2ljZXJ0LmNvbTBLBgNVHR8ERDBCMECgPqA8hjpodHRwOi8vY3JsNC5kaWdpY2Vy
dC5jb20vRGlnaUNlcnRIaWdoQXNzdXJhbmNlRVZSb290Q0EuY3JsMD0GA1UdIAQ2
MDQwMgYEVR0gADAqMCgGCCsGAQUFBwIBFhxodHRwczovL3d3dy5kaWdpY2VydC5j
b20vQ1BTMB0GA1UdDgQWBBQ901Cl1qCt7vNKYApl0yHU+PjWDzAfBgNVHSMEGDAW
gBSxPsNpA/i/RwHUmCYaCALvY2QrwzANBgkqhkiG9w0BAQsFAAOCAQEAnbbQkIbh
hgLtxaDwNBx0wY12zIYKqPBKikLWP8ipTa18CK3mtlC4ohpNiAexKSHc59rGPCHg
4xFJcKx6HQGkyhE6V6t9VypAdP3THYUYUN9XR3WhfVUgLkc3UHKMf4Ib0mKPLQNa
2sPIoc4sUqIAY+tzunHISScjl2SFnjgOrWNoPLpSgVh5oywM395t6zHyuqB8bPEs
1OG9d4Q3A84ytciagRpKkk47RpqF/oOi+Z6Mo8wNXrM9zwR4jxQUezKcxwCmXMS1
oVWNWlZopCJwqjyBcdmdqEU79OX2olHdx3ti6G8MdOu42vi/hw15UJGQmxg7kVkn
8TUoE6smftX3eg==
-----END CERTIFICATE-----
```

Step 2-4:
Below is the value of *n* that is extracted from the x509 certificate using the flag “modulus”. And by printing out all the fields, we found the value of the exponent ‘e’.

Since there was no specific *openssl* command to extract the signature field, we needed to print out all the fields, and copy/paste the specific signature block into a file, but since we needed a hex-string version of the signature, we used the *-d* flag to delete the “:” and spaces from the data.

To take advantage of the ASN.1 encoding used on X.509 certificates, we used the built-in ASN.1 parser in *openssl* called *asn1parse*. Once parsed, we use the starting offset of 4 and *-strparse* flag to only retrieve the body of the certificate without including the signature block. And with the both of the certificate, we calculated the hash, which is notated below as *hashed body*.

```
mod
D753A40451F899A616484B6727AA9349D039ED0CB0B00087F1672886858C8E63
DABCB14038E2D3F5ECA50518B83D3EC5991732EC188CFAF10CA6642185CB0710
34B052882B1F689BD2B18F12B0B3D2E7881F1FEF387754535F80793F2E1AAAA8
1E4B2B0DABB763B935B77D14BC594BDF514AD2A1E20CE29082876AAEEAD764D6
9855E8FDAF1A506C54BC11F2FD4AF29DBB7F0EF4D5BE8E16891255D8C07134EE
F6DC2DECC48725868DD821E4B04D0C89DC392617DDF6D79485D80421709D6F6F
FF5CBA19E145CB5657287E1C0D4157AAB7B827BBB1E4FA2AEF2123751AAD2D9B
86358C9C77B573ADD8942DE4F30C9DEEC14E627E17C0719E2CDEF1F910281933

exp
10001

sig
700f5a96a758e5bf8a9da827982b007f26a907daba7b82544faf69cfbcf25903
2bf2d5745825d81ea42076626029732ad7dccc6f77856bca6d24f83513473fd2
e2690a9d342d7b7b9bcd1e75d5506c3ecb1ca330b1aa9207a93a767645bd7891
c4ce1a9e22e40b89bae68cc17982a3b8d4c0fc1f2ded4d5255412aa83a2cad07
72ae0ad2c667c44f07171899f765a95760155a344c11cff6cf6b213680efc6f1
5463263539eebbc483649b240a73eca0481673c8b9d7485556987af7bb975c69
a406180478dafe9876be222f7f0777874e88199af855ec5c122a5948db493e15
5e675aa25eeecc53288c0e33931403640bc5e5780994015a75fc929dafed7a29

hashed body
85088f934d3e58e3673ea5be32c7c8cf6965e4ab93fbed4fff634723f46d5693
```

Step 5:
Below is finally using the CA's public key/signature and the body of the server's certificate and running our program, the underlines verify that the program works. There is SHA-256 padding that we can ignore. Looking at the last bit of characters we can see that both hashes are indeed the same.

Hash Function	Length (bit)	Octet String
MD2	144	3020300c06082a864886f70d020205000410
MD5	144	3020300c06082a864886f70d020505000410
SHA-1	120	3021300906052b0e03021a05000414 $(= T_{SHA1})$
SHA-256	152	3031300d060960864801650304020105000420
SHA-384	152	3041300d060960864801650304020205000430
SHA-512	152	3051300d060960864801650304020305000440

Magic bytes (padding) for digest algorithms

```
[05/02/19]seed@VM:~/.../hw3$ ./a.out
verify = 01FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF00303
1300D06096086480165030402010500042085088F934D3E58E3673EA5BE32C7C8CF6965E4AB93FBED4FFF634723F46D5693
[05/02/19]seed@VM:~/.../hw3$ ./a.out | tail -c 65
85088F934D3E58E3673EA5BE32C7C8CF6965E4AB93FBED4FFF634723F46D5693
[05/02/19]seed@VM:~/.../hw3$ sha256sum c0_body.bin
85088f934d3e58e3673ea5be32c7c8cf6965e4ab93fbed4fff634723f46d5693  c0_body.bin
```

Code:

```
// Eduard Klimenko
// Textbook RSA (without padding).
#include <stdio.h>
#include <openssl/bn.h>

int main (){
  BN_CTX *ctx = BN_CTX_new();        // struct used to hold BIGNUM
  BIGNUM *one = BN_new();            // the number 1
  BIGNUM *p = BN_new();              // p
  BIGNUM *p_minus_1 = BN_new();      // p-1
  BIGNUM *q = BN_new();              // q
  BIGNUM *q_minus_1 = BN_new();      // q-1
  BIGNUM *n = BN_new();              // n = p*q
  BIGNUM *e = BN_new();              // e
  BIGNUM *e_mult_inverse = BN_new();  // e^-1
  BIGNUM *phi = BN_new();            // phi = (p-1)*(q-1)
  BIGNUM *d = BN_new();              // d
  BIGNUM *m_in = BN_new();           // message to be encrypted
  BIGNUM *m_out = BN_new();          // message produced from decryption
  BIGNUM *c = BN_new();              // ciphertext

  // the number 1
  BN_dec2bn(&one,"1");

  // should be randomly generated
  BN_dec2bn(&p,"7");
  BN_dec2bn(&q,"11");
  BN_dec2bn(&e,"7");

  // message
  BN_dec2bn(&m_in,"25");

  // key generation:
  // n = p*q
  BN_mul(n, p, q, ctx);

  // p-1 and q-1
  BN_sub(p_minus_1, p, one);
  BN_sub(q_minus_1, q, one);

  // phi(n) = (p-1)*(q-1)
  BN_mul(phi, p_minus_1, q_minus_1, ctx);

  // e*x+phi(n)*y=1
  BN_mod_inverse(e_mult_inverse, e, phi, ctx);

  // d = e^-1 mod phi(n)
  BN_mod(d, e_mult_inverse, phi, ctx);

  // print out p, q, e, n, d, m
  printf("p = %s, q = %s\n\n", BN_bn2dec(p), BN_bn2dec(q));
  printf("Public Key: e = %s, n = %s\nPrivate Key: d = %s\n\n",BN_bn2dec(e), BN_bn2dec(n),
BN_bn2dec(d));
  printf("m = %s\n",BN_bn2dec(m_in));

  // encryption: c = m^e mod n
  BN_mod_exp(c, m_in, e, n, ctx);
  printf("c = %s\n\n",BN_bn2dec(c));

  // decryption: m = c^d mod n
  BN_mod_exp(m_out, c, d, n, ctx);
  printf("c decrypted = %s\n",BN_bn2dec(m_out));

  // free struct
  BN_CTX_free(ctx);

  return 0;
}
```

Key Generation

$p = 7, q = 11$
$n = 77 \quad \phi(n) = (7-1)(11-1) = 60$
$e = 7$
$d = 7^{-1} \, mod \, 60 = 43$
$pub: \{7, 77\} \quad priv: \{43, 7, 11\}$

Encrypt
$M = 25$
$C = M^e \, mod \, n = 25^7 mod \, 77 = 53$

Decrypt
$M = C^d \, mod \, n = 53^{43} \, mod \, 77$

$C_p = 53^{43} mod \, 7$
$43 \, mod \, 6 = 1$
$53 \, mod \, 7 = 4$
$C^p = 4^1 mod \, 7 = 4$

$C_q = 53^{43} \, mod \, 11$
$43 \, mod \, 10 = 3$
$53 \, mod \, 11 = 9$
$C_q = 9^3 \, mod \, 11 = 3$

$M = 4(11 * 11^{-1} \, mod \, 7) + 3(7 * 7^{11} \, mod \, 11) \, mod \, 77 = 25$

The program was built off of this small example for correctness as there was no way to verify the algorithm at first. For the completion of the lab, the only variables that were changed were the inputs such as p, n, e, d and the message.