RSA Encryption and Digital Signatures

1 Overview

RSA (Rivest–Shamir–Adleman) is one of the first public-key cryptosystems and is widely used for secure communication. The RSA algorithm is used for encryption, decryption, and digital signatures. The learning objective of this assignment is to gain hands-on experience with RSA.

Readings and videos. Please see the Week 4 course notes for an introduction to public-key cryptography and the RSA cryptosystem. For more detailed coverage of RSA, you may refer to Chapter 7 of *Understanding Cryptography* by C. Paar and J. Pelzl, available at: `https://link-springer-com.proxy.queensu.ca/content/pdf/10.1007/978-3-642-04101-3.pdf`.

2 Submission

Please submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. Please answer the questions in detail, and explain any observations you make that are interesting or surprising. Please also explain any code that you write. Code submitted without any accompanying explanations will result in grade deductions.

Lab environment. This lab has been tested on the SEED Ubuntu 20.04 VM. You can download a pre-built image from the SEED website, and run the SEED VM on your own computer. However, most of the SEED labs can be conducted on the cloud, and you can follow our instruction to create a SEED VM on the cloud. **You may complete this assignment in any Linux-based environment, but any code that you write should be able to compile and run on the provided VM.**

Acknowledgment. The original SEED Lab version of this document was developed with the help of Shatadiya Saha, a graduate student in the Department of Electrical Engineering and Computer Science at Syracuse University.

3 Lab Tasks

The RSA algorithm involves computations on large integers that are typically greater than 1024 bits. In programming languages such as C or Java, these computations cannot be directly programmed using standard arithmetic operators, which are designed to operate on primitive data types such as 32-bit integers or 64-bit long integers. Performing arithmetic operations on arbitrarily large integers requires using additional libraries. For example, OpenSSL provides a Big Number library which can instantiate `BIGNUM` types,

and provides special functions to perform operations such as addition, multiplication, or exponentiation. Appendix A provides an overview of the `BIGNUM` API provided by OpenSSL.

You will need to complete the first five tasks of this assignment in C. But you may complete the final task (cracking RSA keys) using any programming language/library or using any appropriate tools that you can find. However, you must provide a complete description of the methods that you used.

To avoid mistakes, please avoid manually typing the numbers used in the lab tasks. Instead, copy and paste them from this PDF file. **All numerical values are provided in hexadecimal. Please also use hexadecimal when providing any numerical values.**

3.1 Task 1: Deriving the Private Key

Let `p`, `q`, and `e` be three prime numbers. Let `n = p*q`. We will use `(e, n)` as the public key. The hexadecimal values of `p`, `q`, and `e` are listed below. Although the `p` and `q` below are quite large, they are not large enough to be secure. We intentionally make them small for simplicity. In practice, these numbers should be at least 2048 bits long.

```
p = 0x5adb09bdbee704d048da9
q = 0x14738741a1260adf78815
e = 0x10001
```

1. What is the bit length of the modulus `n`?
2. Calculate the private key `d`.

3.2 Task 2: Encrypting a Message

Encrypt the message `i<3crypto` using the public key `(e, n)` from Task 1. You will first need to convert the message from ASCII to hexadecimal, and then convert the hexadecimal number to a BIGNUM using `BN_hex2bn()`.

You may use Python to convert an ASCII string to hexadecimal as follows.

```
>>> import binascii
>>> print(binascii.hexlify(b"kappa").decode("ascii"))
6b61707061
```

3.3 Task 3: Decrypting a Message

Using the private key from Task 1, decrypt the following ciphertext `C` and convert it back to an ASCII string.

```
C = 0x12e6c403952e461afd626e2c81d7b53d0cfe043f3
```

You may use Python to convert a hexadecimal number into ASCII as follows.

```
>>> import binascii
>>> print(binascii.unhexlify(b"6b61707061").decode("ascii"))
kappa
```

3.4 Task 4: Signing a Message

RSA digital signatures use the same RSA algorithm that is used for RSA encryption. In practice, you should always generate separate RSA key pairs for signing and for encrypting. But since this is just for practice, we will use the same key pair from Task 1 and ignore the fact that Tasks 2 and 3 were about encryption and decryption.

1. Generate a signature for the following message (sign this message directly, without hashing it first):

   ```
   I owe you $100
   ```

2. Which exponent did you need to use for signing? (Refer to the course slides.)

3. Change one character in the message and sign the modified message. Compare both signatures and describe what you observe.

3.5 Task 5: Verifying a Signature

Bob receives a message M = `"Launch a missile."` from Alice, with her signature `S`. We know that Alice's public key is `(e, n)`. Please verify whether the signature is indeed Alice's or not. The public key and signature are listed in the following:

```
M = Launch a missile.
S = 0x643d6f34902d9c7ec90cb0b2bca36c47fa37165c0005cab026c0542cbdb6802f
e = 0x010001
n = 0xae1cd4dc432798d933779fbd46c6e1247f0cf1233595113aa51b450f18116115
```

Suppose the signature above is corrupted, such that the last byte of the signature changes from `2F` to `3F`, i.e, only one bit is corrupted. Please repeat this task and describe what happens to the verification process.

3.6 Task 6: Cracking an RSA Private Key

1. Crack the RSA private key corresponding to the public key below. You may want to find some libraries that implement efficient integer factorization algorithms. Cracking this key may take in the order of minutes using single-threaded implementations of factoring algorithms found in popular libraries.

   ```
   e = 0x10001
   n = 0x7c5cfe617c286a27ffc10ecf88a8d35ebbf1e30320af
   ```

2. Crack the RSA private key corresponding to the public key below. Using an appropriate tool/algorithm will allow you to crack this key in a short time (e.g., less than a minute on a 16-core AMD Ryzen 9 5950X processor), but if you use a more basic/general-purpose implementation you may have to wait much longer.

   ```
   e = 0x10001
   n = 0x1e6f1558fe63761406be065a9e07b060f8360b2725b09f4071
   ```

3. So you've made it this far! Your final challenge is to crack the RSA private key corresponding to the public key below. This might take a moderate amount of time if using the appropriate tool for the job, but otherwise it may take an extremely long time (i.e., multiple hours)—so you are better off finding the right tool.

```
e = 0x10001
n = a962e5e6455feec804c6e6faa62b66a50675a8
      24e5e73ac03b009a7d36abb557c14418444d45
```

4. Compare your observations about cracking the three private keys above. Did you start with a more basic method for cracking the first key, before moving to a more advanced method for the second or third? Share how long it took to crack each key on your computer, along with your computer's specifications (CPU model number and RAM). Share your observations on how the cracking time grows with respect to the key length.

Note that this task isn't testing your programming and mathematical skills as much as it is your ability to search online and find the right tools and resources to get the job done. See this lecture by a professor at Carnegie Mellon University talking about "How the Best Hackers Learn Their Craft", to gain an appreciation for why this is important and how it will help you become a better security practitioner or researcher: `https://www.youtube.com/watch?v=6vj96QetfTg`.

A The BIGNUM APIs

All the big number APIs can be found from `https://linux.die.net/man/3/bn`. Below, we describe some of the APIs that we need for this lab.

- Some of the library functions requires temporary variables. Since dynamic memory allocation to create BIGNUMs is quite expensive when used in conjunction with repeated subroutine calls, a `BN_CTX` structure is created to holds BIGNUM temporary variables used by library functions. We need to create such a structure, and pass it to the functions that requires it.

```
BN_CTX *ctx = BN_CTX_new()
```

- Initialize a BIGNUM variable.

```
BIGNUM *a = BN_new()
```

- There are a number of ways to assign a value to a BIGNUM variable.

```
// Assign a value from a decimal number string
BN_dec2bn(&a, "12345678901112231223");

// Assign a value from a hex number string
BN_hex2bn(&a, "2A3B4C55FF77889AED3F");

// Generate a random number of 128 bits
BN_rand(a, 128, 0, 0);

// Generate a random prime number of 128 bits
BN_generate_prime_ex(a, 128, 1, NULL, NULL, NULL);
```

- Print out a big number.

```
void printBN(char *msg, BIGNUM * a)
{
```

```
   // Convert the BIGNUM to number string
   char * number_str = BN_bn2dec(a);

   // Print out the number string
   printf("%s %s\n", msg, number_str);

   // Free the dynamically allocated memory
   OPENSSL_free(number_str);
}
```

- Compute `res` $= a - b$ and `res` $= a + b$:

```
BN_sub(res, a, b);
BN_add(res, a, b);
```

- Compute `res` $= a * b$. It should be noted that a `BN_CTX` structure is need in this API.

```
BN_mul(res, a, b, ctx)
```

- Compute `res` $= a * b$ `mod n`:

```
BN_mod_mul(res, a, b, n, ctx)
```

- Compute `res` $= a^c$ `mod n`:

```
BN_mod_exp(res, a, c, n, ctx)
```

- Compute modular inverse, i.e., given `a`, find `b`, such that $a * b$ `mod n = 1`. The value `b` is called the inverse of `a`, with respect to modular `n`.

```
BN_mod_inverse(b, a, n, ctx);
```

A.1 A Complete Example

We show a complete example in the following. In this example, we initialize three BIGNUM variables, `a`, `b`, and `n`; we then compute $a * b$ and `(`a^b `mod n)`.

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

#define NBITS 256

void printBN(char *msg, BIGNUM * a)
{
   /* Use BN_bn2hex(a) for hex string
    * Use BN_bn2dec(a) for decimal string */
   char * number_str = BN_bn2hex(a);
   printf("%s %s\n", msg, number_str);
   OPENSSL_free(number_str);
}
```

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

  BIGNUM *a = BN_new();
  BIGNUM *b = BN_new();
  BIGNUM *n = BN_new();
  BIGNUM *res = BN_new();

  // Initialize a, b, n
  BN_generate_prime_ex(a, NBITS, 1, NULL, NULL, NULL);
  BN_dec2bn(&b, "273489463796838501848592769467194369268");
  BN_rand(n, NBITS, 0, 0);

  // res = a*b
  BN_mul(res, a, b, ctx);
  printBN("a * b = ", res);

  // res = a^b mod n
  BN_mod_exp(res, a, b, n, ctx);
  printBN("a^c mod n = ", res);

  return 0;
}
```

Compilation. We can use the following command to compile `bn_sample.c` (the character after - is the letter ℓ, not the number 1; it tells the compiler to use the `crypto` library).

```
$ gcc bn_sample.c -lcrypto
```