**CITS 3004**
**Cybersecurity**

THE UNIVERSITY OF WESTERN AUSTRALIA

# Lab 3: Public Key Encryption

## Task 1: RSA

### 1A) BIGNUM API

For this lab, you are welcome to use Python, which does not need the BIGNUM API (big number types are handled automatically). However, Lab 4 will primarily be based on C, so you are requested to complete this task for future references.

The RSA algorithm involves computations on large numbers. These computations cannot be directly conducted using simple arithmetic operators in programs, because those operators can only operate on primitive data types, such as 32-bit integer and 64-bit long integer types. The numbers involved in the RSA algorithms are typically more than 512 bits long. For example, to multiple two 32-bit integer numbers `a` and `b`, we just need to use `a*b` in our program. However, if they are big numbers, we cannot do that anymore; instead, we need to use an algorithm (i.e., a function) to compute their products.

There are several libraries that can perform arithmetic operations on integers of arbitrary size. In this lab, we will use the Big Number library provided by `openssl`. To use this library, we will define each big number as a `BIGNUM` type, and then use the APIs provided by the library for various operations, such as addition, multiplication, exponentiation, modular operations, etc.

All the big number APIs can be found from `https://linux.die.net/man/3/bn`. In the following, we describe some of the APIs that are needed 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()
```

# CITS 3004
# Cybersecurity

THE UNIVERSITY OF WESTERN AUSTRALIA

- 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 needed 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 complete example is shown 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;
}
```

We can use the following command to compile bn_sample.c (the character after – is the letter L in small letters, not number 1; it tells the compiler to use the crypto library).

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

# CITS 3004 Cybersecurity

## 1B) RSA COMPUTATION

### 1B1) 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. Please calculate the private key d. The hexadecimal values of `p`, `q`, and `e` are listed in the following. It should be noted that although `p` and `q` used in this task are quite large numbers, they are not large enough to be secure. We intentionally make them small for the sake of simplicity. In practice, these numbers should be at least 512 bits long (the one used here are only 128 bits).

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

### 1B2) Encrypting a Message

Let `(e, n)` be the public key. Please encrypt the message `"A top secret!"` (the quotations are not included). We need to convert this ASCII string to a hex string, and then convert the hex string to a BIGNUM using the hex-to-bn API `BN_hex2bn()`. The following python 3 command can be used to convert a plain ASCII string to a hex string. Note, encoding and decoding may take a while.

```
$ python -c 'print("A top secret!".encode("utf-8").hex())'
4120746f702073656372657421
```

The public keys are listed in the followings (hexadecimal). We also provide the private key `d` to help you verify your encryption result.

```
n = DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5
e = 010001 (this hex value equals to decimal 65537)
M = A top secret!
d = 74D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D
```

### 1B3) Decrypting a Message

The public/private keys used in this task are the same as the ones used in Task 2. Please decrypt the following ciphertext `C`, and convert it back to a plain ASCII string.

```
C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
```

You can use the following python 3 command to convert a hex string back to a plain ASCII string.

```
$ python -c 'print(bytes.fromhex("4120746f702073656372657421").decode('utf-8'))'
A top secret!
```

### 1B4) Signing a Message

RSA is also widely used for signing messages to ensure the integrity (i.e., the message is not tampered). For example, a simple approach is for the sender to create a signature by encrypting the message `M` using the private key `(d, n)` such that `S(M) = M`$^d$ `mod n`. The sender will send the original message and the signature `(M, S(M))`. The receiver can verify the message received has not been tampered by using the sender's public key `(e, n)` to compute `M = S(M)`$^e$ `mod n`. Of course, the current digital signature procedure involves using the hash, which will be covered in the next section.

The public/private keys used in this task are the same as the ones used in question 1B2. Please generate a signature for the following message (please directly sign this message, instead of signing its hash value):

```
M = I owe you $2000
```

Please make a slight change to the message M, such as changing `$2000` to `$3000`, and sign the modified message. Compare both signatures and describe what you observe.

### 1B5) 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 (hexadecimal) are listed in the following:

```
M = Launch a missle.
S = 643D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F
e = 010001 (this hex value equals to decimal 65537)
n = AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115
```

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

**Answer Q1 ~ Q10 on LMS**

# CITS 3004 Cybersecurity

## Task 2: DH

Written by Jin Hong

### 2A) SETUP

You are welcome to work in peers for this task, but it can still be completed by yourself.

Determine Alice and Bob. The agreed values for the DH scheme are $g=2879$ and $N=9929$. Calculate their shared key given Alice choses her secret value $a=9$ and Bob choses his secret value $b=6$. (i.e., Alice will calculate $g^a \bmod N$ and send it to Bob, etc). Validate that both Alice and Bob calculates the same shared key.

### 2B) MITM ATTACK

DH key exchange scheme is known to be vulnerable to a man-in-the-middle (MITM) attack. Insert Eve to the scenario given in question 2A. Eve's secret value is 5. Calculate the shared keys between (Alice, Eve) and (Eve, Bob).

### 2C) MITIGATING MITM ATTACK

One approach for mitigating a MITM attack is to ensure that Alice and Bob verify their identities. To do so, they must sign their message using the receiver's public value. For example, Alice will send her public value $g^a$, but also her identify $A$. Then, Bob can send back to Alice with his public value $g^b$, along with his identity $B$ and a signature $SIG_B(g^a, g^b, A)$. For this to work, we assume Alice and Bob know each other's identity. Let identity be simple values, for Alice $A=2791$, and Bob $B=8507$. In addition, Eve's identify is $E=4617$. For the signing purposes, we will use SHA-1 hash algorithm (there are standard libraries provided to use this in various languages), where values $g^a$, $g^b$ and $A$ are all concatenated. For example, Bob will sign the value 361448502791 (as a byte string) using the SHA1 hash algorithm. Alice can confirm the received message came from Bob by checking the signature (i.e., Alice also computes the signature for matching).

**Note 1:** Proper digital signatures are used for authenticating identities.
**Note 2:** Currently, the main authenticated key exchange (AKE) protocol used is Sign-and-MAC (SIGMA)[1]. But there are other AKE that can be used, such as MQV, HMQV, CMQV etc.

**Answer Q11 ~ Q15 on LMS**

## References

1. https://en.wikipedia.org/wiki/Proof_of_knowledge#Sigma_protocols