CS 4236 Cryptography Theory and Practice
Assignment 2

(Due: 11:59 PM (SGT), 20 October 2019)

Student ID: **DON'T FORGET YOUR STUDENT ID**

Submission instructions:

- Please submit assignment via LumiNUS.
- Please include your *answers only*, *not* the questions.
- Please use proper text editors or LaTeX. (No handwritten answers!)
- Remember that you have three grace days for the entire semester.

Part I: Crypto Library Exercise

1. Visit the following site: http://www.cis.syr.edu/~wedu/seed/lab_env.html and download `SEEDUbuntu-16.04-32bit.zip` folder.
2. Follow the the documents: `SEEDVMVirtualBoxManual.pdf` and `Ubuntu1604VMManual.pdf` (both found in the above website) to setup your environment.

Note that this submission requirement is for Part I only. In your report, you should **explain your answer and include screenshots** of the commands/outputs wherever applicable.
To avoid mistakes, please avoid manually typing the numbers. Instead, copy and paste the numbers from this PDF file.

1 Secret Key Encryption (20 points)

1.1 Encryption using Different Ciphers and Modes

In this task, we will play with various encryption algorithms and modes. You can use the following `openssl enc` command to encrypt/decrypt a file. To see the manuals, you can type `man openssl` and `man enc`.

```
$ openssl enc ciphertype -e  -in plain.txt -out cipher.bin \
             -K  00112233445566778889aabbccddeeff \
             -iv 0102030405060708
```

Please replace the `ciphertype` with a specific cipher type, such as `-aes-128-cbc`, `-aes-128-cfb`, `-bf-cbc`, etc. In this task, you should try at least 3 different ciphers and three different modes. Take a screenshot of the commands used for various modes of encryption. You can find the meaning of the command-line options and all the supported cipher types by typing `"man enc"`. We include some common options for the `openssl enc` command in the following:

```
-in <file>     input file
-out <file>    output file
-e             encrypt
-d             decrypt
-K/-iv         key/iv in hex is the next argument
-[pP]          print the iv/key (then exit if -P)
```

1.2 Encryption Mode – ECB vs. CBC

The file `pic_original.bmp` can be found under `./picture/`, and it contains a simple picture. We would like to encrypt this picture, so people without the encryption keys cannot know what is in the picture. Please encrypt the file using the ECB (Electronic Code Book) and CBC (Cipher Block Chaining) modes, and then do the following:

1. Let us treat the encrypted picture as a picture, and use a picture viewing software to display it. However, for the .bmp file, the first 54 bytes contain the header information about the picture, we have to set it correctly, so the encrypted file can be treated as a legitimate `.bmp` file. We will replace the header of the encrypted picture with that of the original picture. We can use the `bless hex` editor tool (already installed on our VM) to directly modify binary files. We can also use the following commands to get the header from `p1.bmp`, the data from `p2.bmp` (from offset 55 to the end of the file), and then combine the header and data together into a new file.

   ```
   $ head -c 54 p1.bmp > header
   $ tail -c +55 p2.bmp > body
   $ cat header body > new.bmp
   ```

2. Display the encrypted picture using a picture viewing program (we have installed an image viewer program called eog on our VM). Take a screenshot of both the encrypted picture. Can you derive any useful information about the original picture from the encrypted picture? Please explain your observations.

1.3 Initialization Vector (IV)

The objective of this task is to help students understand the problems if an IV is not selected properly. Please do the following experiments:

1. A basic requirement for IV is uniqueness, which means that no IV may be reused under the same key. To understand why, please encrypt the same plaintext using (1) two different IVs, and (2) the same IV. You may use *diff*

command to display difference between two binary files. Take a screenshot of the commands and outputs that explains your answer. Please describe your observation, based on which, explain why IV needs to be unique.

2. One may argue that if the plaintext does not repeat, using the same IV is safe. Let us look at the Output Feedback (OFB) mode. Assume that the attacker gets hold of a plaintext (P1) and a ciphertext (C1) , can he/she decrypt other encrypted messages if the IV is always the same? You are given the following information, please try to figure out the actual content of P2 based on C2, P1, and C1. Take a screenshot of the commands and outputs that explains your answer.

```
Plaintext  (P1): This is a known message!
Ciphertext (C1): a469b1c502c1cab966965e50425438e1bb1b5f9037a4c159
Plaintext  (P2): (unknown to you)
Ciphertext (C2): bf73bcd3509299d566c35b5d450337e1bb175f903fafc159
```

   If we replace OFB in this experiment with CFB (Cipher Feedback), how much of P2 can be revealed? You only need to answer the question; there is no need to demonstrate that.

3. From the previous tasks, we now know that IVs cannot repeat. Another important requirement on IV is that IVs need to be unpredictable for many schemes, i.e., IVs need to be randomly generated. In this task, we will see what is going to happen if IVs are predictable.

   Assume that Bob just sent out an encrypted message, and Eve knows that its content is either Yes or No; Eve can see the ciphertext and the IV used to encrypt the message, but since the encryption algorithm AES is quite strong, Eve has no idea what the actual content is. However, since Bob uses predictable IVs, Eve knows exactly what IV Bob is going to use next. The following summarizes what Bob and Eve know:

```
Encryption method: 128-bit AES with CBC mode.
  Key (in hex):00112233445566778899aabbccddeeff (known only to Bob)
  Ciphertext (C1): bef65565572ccee2a9f9553154ed9498 (known to both)
  IV used on P1 (known to both):
  (in ascii): 1234567890123456
  (in hex) : 31323334353637383930313233343536
  Next IV (known to both)
  (in ascii): 1234567890123457
  (in hex)  : 31323334353637383930313233343537
```

   Your job is to construct a message P2 and ask Bob to encrypt it and give you the ciphertext. Your objective is to use this opportunity to figure out whether the actual content of P1 is Yes or No. Please take a screenshot of the commands and outputs that explains your answer.

2 Public Key Encryption (30 points)

2.1 Background

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 any more; 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.

2.2 BIGNUM APIs

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()
```

- 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);
}
```

- Computations

```
1. Compute res = a - b and res = a + b:
BN_sub(res, a, b);
BN_add(res, a, b);

2. Compute res = a * b. It should be noted that a BN_CTX
structure is need in this API
BN_mul(res, a, b, ctx)

3. Compute res = a * b  mod n:
BN_mod_mul(res, a, b, n, ctx)

4. Compute res = a^c  mod n:
BN_mod_exp(res, a, c, n, ctx)

5. 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);
```

- Complete Example

```
/* 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 and execution. 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
$ ./a.out
```

2.3 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
```

2.4 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 command can be used to convert a plain ASCII string to a hex string.

```
$python -c 'print("A top secret!".encode("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
```

2.5 Decrypting a Message

The public/private keys used in this task are the same as the ones used in section 2.4. Please decrypt the following ciphertext C, and convert it back to a plain ASCII string.

```
C = 8C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F
```

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

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

2.6 Signing a Message

The public/private keys used in this task are the same as the ones used in section 2.4. 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.

2.7 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 describe what will happen to the verification process.

Part II: Creating Your Own Padding Oracle Attacks

Part II is about attacking a cipher using a padding oracle. For this task, you will implement the padding attack discussed in class.

3 Padding Oracle Attack (30 points)

You will have the opportunity to create your own padding attack discussed in class to extract a message from a ciphertext. You can find a set of valid ciphertexts under `./ciphertext/`. The ciphertexts will be given as a 128-bit hexadecimal number (starting with 0x...) computed as follows:

$C = c_0 \| c_1 = CBC_K^{Encrypt}(PAD(M))$,

where the word M is shorter than or equal to 8 ASCII[1] characters (i.e., 64 bits). The block size of the cipher and the length of the secret key K are 64 bits. The first 64-bit block of C is the initialization vector (IV), which we denote c_0, and the last 64-bit block is the first cipher block, c_1. Note that we assume the randomized CBC encryption; that is, the IV (c_0) is randomly selected for every message.

You are given a *padding oracle*, which can be accessed by function calls; see Section [?] for the details. You can access the padding oracle as many times as you want. You will have two language options to develop your padding attack program: Java or Python. The padding oracle will accept both c_0 and c_1 (each 64 bits long), and return a 1 or a 0, indicating correct or incorrect padding respectively. The padding oracle works as follows:

$\{0, 1\} = Check\_PAD(CBC_K^{Decrypt}(C'))$, where you provide the C'; i.e., you are a chosen-ciphertext attacker.

3.1 How to Access the Oracle?

You will be given the following list of files under `./oracle/`:

- `pad_oracle.jar`: Padding oracle (Java bytecode[2]) for Problem 3.
- `dec_oracle.jar`: Decryption oracle (Java bytecode) for Problem 4.
- `oracle_python.py`: Padding and decryption oracles (Python functions) for Problem 3 and 4, respectively.
- `python_interface.jar`: Java bytecode for interfacing Python scripts and oracle Java bytecodes.
- `bcprov-jdk15-130.jar`: Crypto library.

Java:

```
// query padding oracle
pad oracle p = new pad_oracle();
boolean isPaddingCorrect = p.doOracle("0x1234567890abcdef", "0x1234567890abcdef");
// query decryption oracle
dec_oracle d = new dec_oracle();
String plaintext = d.doOracle("0x1234567890abcdef", "0x1234567890abcdef");
```

[1] 1https://en.wikipedia.org/wiki/ASCII#ASCII_printable_code_chart

[2] Note that (1) you should not reverse-engineer the bytecodes and (2) even if you did, it would not help.

Python: Execute the Java-Python interface class by

```
java -cp pad_oracle.jar:dec_oracle.jar:bcprov-jdk15-130.jar:python_interface.jar python_interface
```

In your Python script,

```
from oracle_python import pad_oracle, dec_oracle
# query padding oracle
ret_pad = pad_oracle('0x1234567890abcdef', '0x1234567890abcdef');
# query decryption oracle
ret_dec = dec_oracle('0x1234567890abcdef', '0x1234567890abcdef');
```

3.2 Submission

Your code has name `p3_#.java`(or `.py`), where `#` denotes the student id. Your code should accept two command-line arguments `c_0` and `c_1` in the following format:

- `javac -cp pad_oracle.jar p3_#.java`
  `java -cp pad_oracle.jar:bcprov-jdk15-130.jar:.  p3_# c_0 c_1`
- `python p3_#.py c_0 c_1`

The ciphertext blocks (`c_0` and `c_1`) have hex format starting with 0x. The output must be the plaintext in ASCII format.

4 Turning Decryption Oracle into Encryption Oracle (20 points)

What you will create in Problem 3 is a single-block decryption oracle that takes a two-block ciphertext (one of which is an IV) and returns a single-block plaintext. Now, you are asked to create an *encryption oracle* using the the single-block decryption oracle. Your code needs to take an arbitrary message M, pad it if needed, and encrypt it without knowing the secret key. Note that the message M can be longer than one plaintext block.

We provide a single-block decryption oracle so that you can solve Problem 4 without solving Problem 3. The single-block decryption oracle is accessed via calling the oracle function. Note that the secret key used in Problem 4 is different from the one used in Problem 3.[3]

4.1 Submission

Your code has name `p4_#.java`(or `.py`), where `#` denotes the student id. Your code should accept any arbitrary message M in the command-line argument:

- `javac -cp dec_oracle.jar p4_#.java`
  `java -cp dec_oracle.jar:bcprov-jdk15-130.jar:.  p4_# "This is the message that needs to be encrypted."`
- `python p4_#.py "This is the message that needs to be encrypted."`

The output must be the one or more 64-bit hex formats; e.g., `0x12...ef 0x98..ab 0xb7..2d`.

[3]Since the two keys are different, you cannot use the decryption oracle provided in Problem 4 to solve Problem 3.