Student Name: Yi-Ting, Hsieh(a1691807)

- **Part 1**

From the source code and the description provided from myUni, we learn that the address of our assignment3 is somewhere in the ASCII plaintext, which has been ***encrypted by XORing with the "same randomly generated key pad"***, then encoded in hex. Apparently, it is a one-time pad(OTP) key been used for multiple times, which leads to a vulnerability for us to perform many-time pad(MTP) attack. Keys in OTP should only ne used once.

Reference: MTP(https://github.com/CameronLonsdale/MTP)

First, we use pip to install this software for our MTP attack(notice that it only supports for Python3.7). Then, we store the cipher text to a file called cipher.txt. Now, it is the time to perform MTP attack by calling following command:

```
$ mtp cipher.text
```

Now we can just fill up the reasonable text. We provide a quote here for line 5:
"If someone steals your password, you can change it. But if someone steals your thumbprint, you can't get a new thumb. The failure modes are very different. - Bruce Schneier"

The address of our assignment 3 is here:
"Congratulations on breaking the code.", " Your assignment is in https://cs.adelaide.edu.au/~yval/SP18/A3ishere.php"

The decrypted plain text is provided as *result.json.*

- **Part 2**

1.) Edit our calc program, so that it can call both bn library and ybn library and store the value in separate stack at the same time. In order to achieve this, we have to modify the header of your ybn.h file:
It was:

```
#ifndef __BN_H__
#define __BN_H__ 1
```

We change it to the following to solve the naming conflict:

```
#ifndef __YBN_H__
#define __YBN_H__ 1
```

Every times when user inputs a number or performs an arithmetic operation, we compare the output of our bn and ybn. If the output is different, we call abort to exit the program. The fuzzer would detects the abort signal and cause a crash, which we can examinee and test the crash input and find out the cause of it later.

Under the a1691807/part-2/fuzz folder, contains our initial buggy ybn library with the libbn.a and our calc program:

- Modify makefile(from the tutorial one):
  change to CC=cang-7
  change to AFLCC := afl-clang
- Use following command to build to build our fuzzer:
  ```
  $ make
  ```
- Prepare some simple test cases.
- Command line used to find out the crash:
  ```
  ./calc_fuzz ../testDir/fromstring/
  ./calc_fuzz ../testDir/add/
  ./calc_fuzz ../testDir/sub/
  ./calc_fuzz ../testDir/mul/
  ./calc_fuzz ../testDir/sqr/
  ./calc_fuzz ../testDir/modexp/
  ```

The configure files with the original ybn library are in part-2/fuzz/, which contains

- The calc program.
- The original buggy ybn library
- The libbn.a I used for my assignment2 with bn.c.
- One makefile for libfuzzer.

2.) Use libfuzzer to find and analyzee the cause of the crash or other vulnerabilities of the program.

Found:

ybn:

- `ybn_fromString()`: found crash.
  The ybn doesn’t work for some huge input. Such as 65536.

  ```
  Result:
  =======Detected Different outcome===========
  Bn is: 65536
  YBN is: 0
  ```

The problem is the following section of code in fromString:

```
     int factor =1;
     …
          fact *= 10;
          if (factor == 100000) {...}
```

We are not sure what is the MAX_INT for the environment. In addition, in ybn_mul_add(ybn_t bn, **int factor**, int add), we pass the factor value into this function. Then, the ybn_mul_add perform operation directly without checking the boundary, which may overflow the uint16_t data variable. We can solve the problem by changing to the following line:

```
     if (factor >= 1000) {...}
```

- ybn_add(): pass test.
- ybn_sub(): pass test.
- ybn_mul(): pass test.
- ybn_toString(): pass test.
- ybn_div(): pass test.
- ybn_sqr(): found crash—---evil

The ybn_sqr worked quite good for most of cases, but I found it crashes for some huge input case:(if the ybn_fromstring is working correctly)
1109823928170187903749127390481793847981379847981 sqr

```
Result:
=======Detected Different outcome===========
Bn is:
12317091515391063997028488967363107705989688639951982534295113678719115537971887518372106697
76361
YBN is:
12317091510417831632930622545809285224521480238947421026321636504314475769040292548246772942
43305
```

The problem is the following of the code in ybn_sqr():

```
for (int i = 0; i < alen; i++) {
  uint64_t carry = rd[i*2] + (uint64_t)ad[i] * ad[i];
  rd[i * 2] = carry & 0xFFFFULL;
  carry >>= 16;
  for (int j = i + 1; j < alen; j++) {
      carry += 2 * (uint64_t)ad[i] * ad[j] + rd[i+j];
      rd[i+j] = carry & 0xFFFFULL;
      carry >>= 16;
  }
  carry += rd[i+alen];
  rd[i+alen] = carry & 0xFFFF;
}
```

We can utilize ybn_mul() for this square function:

```
    int ybn_sqr(ybn_t result, ybn_t a) {
       return ybn_mul(result, a, a);
     }
```

- **ybn_modexp**(): found crash.

  The ybn_modexp worked for some cases, if the fromString() and ybn_sqr() are already fixed. However, it crashes quite frequently:
  crash case:

  ```
  1893740198 10983413987 389 modexp
  ```

  Result:

  ```
  =======Detected Different outcome===========
  Bn is: 39
  YBN is: 38
  Aborted
  ```

  The problem is the following of the code in ybn_modexp():

  ```
  for (int i = ybn_reallen(exp); i-- > 0; ) {
    for (int j = 15; j-- > 0;) {
        ybn_sqr(tmp, result);
        ybn_div(quot, result, tmp, modulus);
        if (exp->ybn_data[i] & (1 << j)) {
              ybn_mul(tmp, result, base);
              ybn_div(quot, result, tmp, modulus);
        }
     }
  }
  ```

  In right-to-left algorithm, it should be:

  ```
        base := sqr(base) mod modulus
  ```

  we can solve it by implementing right-to-left binary method for modexp.

  Solution 1:

  We can change the for loop to

  ```
        for(int j = 16; j--; j>0)
  or    for(int j=15; j--; j>= 0)
  ```

  Solution 2:

  ```
  if (copyYBN(tmp, base) == -1) return -1;
  ybn_div(quot, base, tmp, modulus);
  while (exp->ybn_len > 0) {
        if ((exp->ybn_data[0] & 0x0001) == 0x0001) {
              ybn_mul(tmp, result, base);
              ybn_div(quot, result, tmp, modulus);
        }
        ybn_srli(exp, exp, 0x01, 1);
        ybn_sqr(tmp, base);
        ybn_div(quot, base, tmp, modulus);
  }
  ```

  Where the ybn_srli(exp,exp, 0x01, 1) means:

  ```
        exp := exp >> 1
  ```

3.) The fixed ybn library is under the a1691807/part-2/ybn folder. You can also enter the a1691807/part-2 and execute the following command:

```
$ make
```

to build the fuzzer program to test the fixed ybn library.