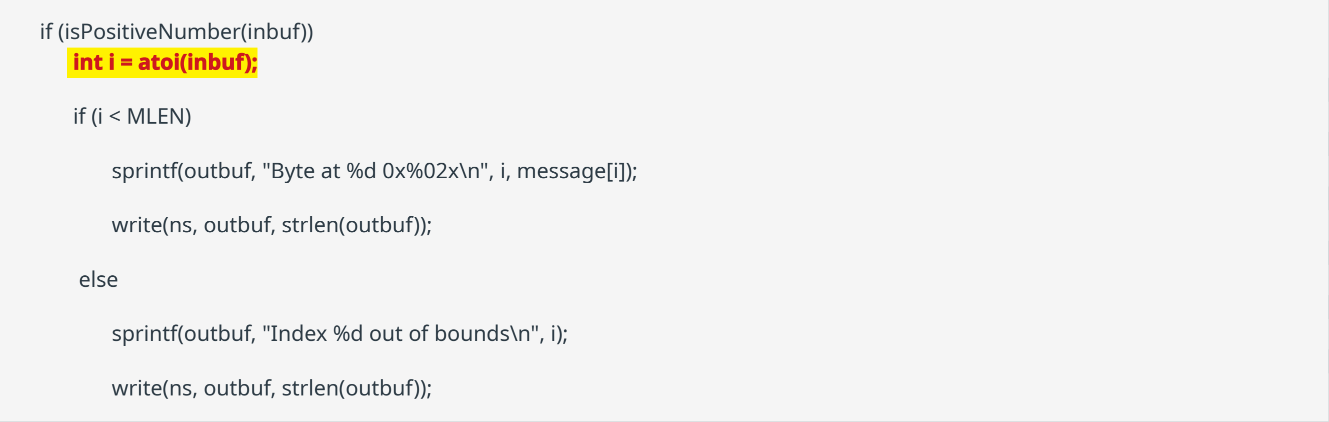
Student Name: Yi-Ting, Hsieh(a1691807)

- **Part 1**

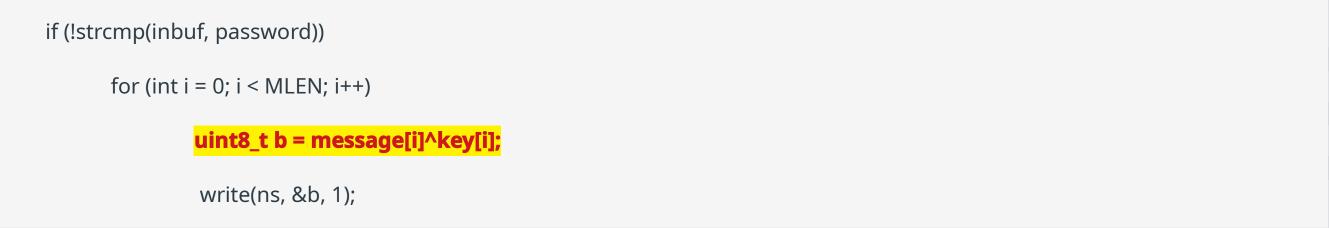
From the source code provided from myUni, we learn that there is a vulnerability for such program. That is in the following code:



```
if (isPositiveNumber(inbuf))
    int i = atoi(inbuf);

    if (i < MLEN)

         sprintf(outbuf, "Byte at %d 0x%02x\n", i, message[i]);

         write(ns, outbuf, strlen(outbuf));

     else

         sprintf(outbuf, "Index %d out of bounds\n", i);

         write(ns, outbuf, strlen(outbuf));
```

Because the program calls isPositiveNumber() to check whether the input only contains digits or not. If so, the program calls atoi function directly without checking the boundary. Then, it checks whether the value return from atoi() is less than MLEN. After a few attempting, we get the MLEN is equal to 54. Also, we get the value of our message.

On the other hand, if we enter the right password, the program will execute the following code:



```
if (!strcmp(inbuf, password))

          for (int i = 0; i < MLEN; i++)

                    uint8_t b = message[i]^key[i];

                    write(ns, &b, 1);
```

Now we know the message can be decrypted by applying XOR operation to corresponding key[i] with message[i], we can learn that this is a stream cipher with symmetric key for both encryption and decryption. So all we have to do is to find out the key.

The program calls atoi without checking the boundary of the input value, we can exploit this to cheat the program so that the atoi function can return a negative number. Because a negative number is always less than MLEN, we can use it to print out some memory blocks, which stored the key values, before the message. Since computer represents signed integers using two's complement, we can enter some positive values which are going to be interpreted as negative number in atoi(). After few attempts, we find out the boundary is near 21474836479. If we input some values that are slightly smaller than this value, we will be able to print out the key.

After getting the key, we perform the XOR operation on our message to corresponding key and get the URL address, which is:

https://cs.adelaide.edu.au/~yval/SP18/assignment2.pdf

- **Part 2**

1.)

```
int32_t f1(int32_t a){
      return a & -a;
}
```

Ans: The function compares the input with the two’s complement of input and returns the value(in int32_t type) of least significant bit which is not 0 in a.

2.)

```
int32_t f2(uint16_t a, uint16_t b){
      return ((int32_t)a - (int32_t)b) >> 31;
}
```

Ans: The function compares the input a and b:
      if a >= b, it returns 0;
      else(when a < b), it returns -1.

3.)

```
int32_t f3(int32_t a){
      return (a | -a) >> 31;
}
```

Ans: The function checks whether the input a is 0,
      if the input a is 0, returns 0;
      else (input a != 0), the function returns -1.

4.)

```
int32_t f4(uint32_t a, uint32_t b, int32_t c, int32_t d){
      c ^= d;
      c = (c | -c) >> 31;
      return (a & ~c) | (b & c);
}
```

Ans:
If input c and input d have different signed, the function returns the int32_t type of b. (if the uint32_t b is >= 2^31, then the int32_t of b will be the some negative number).

else if input c and d have the same signed, the function returns the int32_t type of a. (if the uint32_t a is >= 2^31, then the int32_t a will be some negative number)

p.s) if input c and d are both >= 0, or they are all < 0, then we say c and d have the same signed.

- **Part 3**

1.) Use american fuzzy lop(http://lcamtuf.coredump.cx/afl/) to find out the possible test cases that cause crash.

- Modify makefile:

  change to CC=afl-gcc -fno-stack-protector -z execstack
- Prepare some simple test cases.
- Command line used to find out the crash:
  ```
  ./afl-fuzz -i somePath/testDir/ -o Result/Res_BigNum-# afl_fuzzy_folder/BigNum-#/calc -tmn
  ```

The configure files are in part-3/afl-fuzzy-folder/, which contains

- Different candidates programs file in BigNum-# folder
- Some test cases in testDir/
- The result of afl in Result/Res_BigNum-#

2.) Use libfuzzer to analyze the cause of the crash or other vulnerabilities of the program.

- Modify the makefile, bn.c, calc.c and some other source file for compiling the libfuzzer program.
- Prepare some test cases.
- Use *calc_afl_asan*, *calc_afl, calc_msan, calc_ubsan* to run our program.
- Use above sanitized programs to analyze the test cases we found that caused crash earlier from afl.
- Use above sanitized programs to run some test of our sample test cases.

Student id: a1691**807**

candidates:

BigNum-7:

- dump doesn't work
- swap doesn't work
- Detect memory leak
- Detect stack-buffer-underflow:
  ```
  =24076==ERROR: AddressSanitizer: stack-buffer-underflow on address 0x7ffd5e5f2d38 at pc 0x55ed35d04962 bp 0x7ffd5e5f2b90 sp 0x7ffd5e5f2b88
            calc.c:94:36 in dump
  ==22218==ERROR: AddressSanitizer: stack-buffer-underflow on address 0x7fff48decbf8 at pc 0x55c6bfa12f3d bp 0x7fff48decba0 sp 0x7fff48decb98
            calc.c:25:12 in pop
  ```
- Has Limited Stack size.
- Has Input size limit

BigNum-0:

- Detect memory leak
- Detect heap-buffer-overflow:

  ```
  ==12113==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000003b72 at pc 0x56361074a99e bp 0x7ffee5bce630 sp 0x7ffee5bce628
  ```

  bn_reallen(): bn.c:112:19

  ```
  ==17228==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x607000000070 at pc 0x55f3da85b645 bp 0x7ffdb4d45030 sp 0x7ffdb4d45028
  ```

  calc.c:56:30 in stack_push

BigNum-8:

- Detect memory leak
- Detect heap-buffer-overflow:

  ```
  ==27906==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x602000004c12 at pc 0x564d53daf239 bp 0x7fff15f977a0 sp 0x7fff15f97798
  ```

  bn_sub: bn.c:243:28

  ```
  ==32475==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60300000f1a8 at pc 0x55ee342971ae bp 0x7ffc376058f0 sp 0x7ffc376050a0
  ```

  in _asan_memcpy
- Detect dynamic-stack-buffer-overflow:

  ```
  ==29132==ERROR: AddressSanitizer: dynamic-stack-buffer-overflow on address 0x7ffdd6e10f20 at pc 0x55f7b44eabfe bp 0x7ffdd6e10ef0 sp 0x7ffdd6e10ee8
  ```

  bn_mul: bn.c:320:29

- **ERROR: AddressSanitizer: memcpy-param-overlap**

  ```
  32478==ERROR: AddressSanitizer: memcpy-param-overlap: memory ranges [0x603000000040,0x603000000080) and [0x603000000010, 0x603000000050) overlap
  ```
- Has Input size limit