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, the program 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], also 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 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 signs, then the function returns the int32_t type of b. (if the uint32_t b is >= 2^31, then the int32_t b will be the 2’s complement of b, which will be a negative number)
else if the input c and d have the same sign, the function returns the int32_t type of a. (if the uint32_t a is >= 2^31, then the int32_t a will be the 2’s complement of a, which will be a negative number)

- **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:

  calc.c:94:36 in dump

  calc.c:25:12 in pop
- Has Limited Stack size.

BigNum-0:

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

  bn_reallen(): bn.c:112:19

  calc.c:56:30 in stack_push

BigNum-8:

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

  bn_sub: bn.c:243:28

in asan_memcpy

- Detect dynamic-stack-buffer-overflow:

  bn_mul: bn.c:320:29

- **ERROR: AddressSanitizer: memcpy-param-overlap**
- **Has Input size limit**