CS 161 Project 1: RSA

Due on September 29 at 3:00 PM.

Introduction

In this project you will write a C program that implements textbook RSA encryption and decryption. You will also write code to generate RSA keys.

Why do we say "textbook" RSA? The RSA function alone is not enough to make a secure cryptosystem. For instance, it needs padding to prevent it from being malleable, and it needs randomization so that two encryptions of the same message are not identical. We will not be implementing these other parts of a cryptosystem, only the basic RSA function.

We've provided you with a source code outline and test cases. Your job is to fill in the missing functions to make the tests pass, and answer some written questions. There are three source files: main.c, rsa.c, and rsa.h. You do not have to modify rsa.h. The places in the other two files where you have to write code are marked with `/* TODO */` comments.

The program you will write is called "rsa". It has a command line interface with three modes of execution:

`./rsa encrypt` *`key.pub message`*
: Reads a public key *K* from the file *key.pub*, converts the string *message* to an integer *m*, and outputs the integer representation of $E_K(m)$.

`./rsa decrypt` *`key.priv c`*
: Reads a private key *K* from the file *key.priv* and outputs a string representation of the decryption $D_K(c)$.

`./rsa genkey` *`numbits`*
: Outputs a private key (*d*, *e*, *n*), where *n* is approximately *numbits* bits long. You can store the key to a file using Unix shell redirection: `./rsa genkey 1024 > key.priv`

Sample outputs:

```
$ ./rsa encrypt testkey.pub "hello"
238301160525192269566546305624948574103
$ ./rsa decrypt testkey.priv 238301160525192269566546305624948574103
hello
$ ./rsa genkey 128
d 278503228302265047040587529645952279677
e 65537
n 288614444320069960785732511900748000413
```

Step 0. Get things compiling with GMP

We'll need a way to manipulate large integers. In this project, we'll use the GMP library (https://gmplib.org/). There is documentation for the library online. Start by reading at least the "Basics" section.

https://gmplib.org/manual/
https://gmplib.org/manual/GMP-Basics.html

To install GMP in your home directory on a Hive machine, run these commands:

```
wget https://gmplib.org/download/gmp/gmp-6.0.0a.tar.bz2
tar xvf gmp-6.0.0a.tar.bz2
cd gmp-6.0.0
./configure --prefix=$HOME/gmp
make
make install
```

Copy the file cs161-proj1.tar.gz to a Hive machine. Log in to Hive and extract the files:

```
tar xvf cs161-proj1.tar.gz
```

This will create a directory called cs161-proj1. Enter the directory:

```
cd cs161-proj1
```

To compile the project code:

```
make
```

Any time you change a file, run `make` again. It will notice what has changed and run the appropriate commands to bring everything up to date.

To run the test script:

```
./test.sh
```

You'll notice that even though the program runs, most of the tests fail. You can track your progress by watching the tests begin to pass.

Edit the `main` function in `main.c` and experiment a bit with GMP.

```
    mpz_t a, b, c;

    mpz_init(a);
    mpz_init(b);
    mpz_init(c);

    mpz_set_str(a, "112233445566778899", 10);
    mpz_set_str(b, "998877665544332211", 10);
    /* c = a * b */
    mpz_mul(c, a, b);
    gmp_printf("%Zd = %Zd * %Zd\n", c, a, b);

    mpz_clear(a);
    mpz_clear(b);
    mpz_clear(c);
```

The above code should print:

```
112107482103740987777903741240815689 = 112233445566778899 *
998877665544332211
```

If you're having trouble with GMP, you can check your results using Python, which has built-in large integers (the "L" suffix on the result indicates a large integer):

```
    >>> 112233445566778899 * 998877665544332211
    112107482103740987777903741240815689L
```

Other GMP functions you may find useful:

https://gmplib.org/manual/Integer-Comparisons.html
`mpz_cmp`: comparison

https://gmplib.org/manual/Integer-Exponentiation.html
`mpz_powm`: modular exponentiation

https://gmplib.org/manual/Number-Theoretic-Functions.html
`mpz_probab_prime_p`: test for primality
`mpz_gcdext`: extended GCD
`mpz_invert`: modular inverse

https://gmplib.org/manual/Integer-Import-and-Export.html
`mpz_import`: convert a byte array to an integer

Step 1. Implement the encrypt and decrypt modes

You will implement the `encrypt_mode` and `decrypt_mode` functions (`main.c`), and the `rsa_encrypt` and `rsa_decrypt` functions (`rsa.c`) they depend on.

The input and output of the RSA function are integers mod *n*. But we would like to be able to encrypt strings to get integer ciphertexts and decrypt integer ciphertexts to get strings. The functions `message_encode` and `message_decode` handle these conversions.

In `encrypt_mode`, you should follow this rough outline:

- Call `rsa_key_init` to initialize a key structure.
- Call `rsa_key_load_public` to load the key from a file.
- Convert *message* to an integer *m* using the encode function.
- Call `rsa_encrypt` and output the result.
- Call `rsa_key_clear` to free the public key.

In `decrypt_mode`, you should follow this rough outline:

- Call `rsa_key_init` to initialize a key structure.
- Call `rsa_key_load_private` to load the key from a file.
- Parse the ciphertext string into an integer *c*.
- Call `rsa_decrypt` and store the result in *m*.
- Convert *m* to a string and output it.
- Call `rsa_key_clear` to free the private key.

Run `./test.sh` each time you have implemented new functions.

While you are debugging, it's likely that your decryptions will be incorrect and the decrypt subcommand will output binary gibberish (possibly messing up your terminal state). To prevent this, and make the output easier to read, you can pipe the output into `hexdump`:

```
$ ./rsa decrypt testkey.priv 238301160525192269566546305624948574103 | hexdump -C
00000000  68 65 6c 6c 6f                                    |hello|
00000005
$ ./rsa decrypt testkey.priv 238301160525192269566546305624948574104 | hexdump -C
00000000  ac 15 34 f5 b6 51 c2 75  71 f6 f1 f3 3d ff d1 8a  |..4..Q.uq...=...|
00000010
```

Step 2. Implement the genkey mode

So far we have been using the pre-generated keys that came with the source code. Now we will generate our own keys. The functions you need to implement are `genkey_mode`, `rsa_genkey`, and `generate_prime`.

For RSA, we need to generate two prime numbers. How do we generate prime numbers? We just generate random integers and test them for primality. The prime numbers are dense enough among integers that you can just repeat the process until you get a prime.

Where do we get random numbers from? You may have noticed that GMP has functions such as `gmp_randinit_default` and `mpz_urandomb`. We will **not** use these functions, as they are not random number generators that are suitable for cryptography. Instead, we will ask for randomness straight from the operating system, using the Unix `/dev/urandom` interface.

You have to write the `generate_prime` function. A rough outline is provided:

- Allocate an array of *numbits*/8 bytes using malloc. (You can assume that *numbits* is a multiple of 8.)
- Open a file handle on `/dev/urandom`.
- Read *numbits*/8 bytes from the file handle into the array.
- Set the top two bits of the first byte in the array, to ensure the integer is large enough. In C, you can set the top two bits of a byte variable `b` with: `b = b | 0xc0`.
- Call the `mpz_import` function to convert the byte array to an integer.
- Test the integer using `mpz_probab_prime_p`.
- If it is not prime, go back to the "Read *numbits*/8 bytes" step and try again.
- Free the byte array using `free`.
- Close the `/dev/urandom` file handle.

Read about the possible return values of `fopen` and `fread`. If an error occurs while reading from `/dev/urandom`, do not continue processing. It is better to crash the program (by calling abort, for example) than to continue with possibly bad randomness.

Generate *p* and *q* with half your desired number of bits, so that their product will have the desired number of bits. You can assume that the *numbits* argument to `rsa_genkey` is a multiple of 16 (so that the arguments to `generate_prime` are a multiple of 8). That is, it's fine to abort the program if that precondition does not hold.

With the primes *p* and *q* in hand, you need to derive *d* and *e*. We know that *e* has to be coprime to $(p-1)(q-1)$ because it has to have a multiplicative inverse in a group of that order. So we can actually just take *e* to be a fixed prime number (because a prime number is coprime to all integers that are not a multiple of it). Taking $e=65537$ is a traditional choice. Now all that remains is to compute $d=e^{-1} \bmod (p-1)(q-1)$. You can do that with the `mpz_invert` function.

In `genkey_mode`, you should follow this rough outline:

- Call `rsa_key_init` to initialize a key.
- Call `rsa_genkey`.
- Call `rsa_write(stdout, &key)` to output the key.

Run `./test.sh` again and make sure everything passes.

Concluding Questions

In a separate file called answers.txt, please put down the answers to the following questions:

1. What happens when you try to encrypt a very long message (one that is longer than *n* bits after calling `message_encode`)? Why does that happen? How do cryptosystems that use RSA encrypt inputs of arbitrary length in practice?
2. Encrypt a message with the test public key. Now encrypt the exact same message again. What do you notice? Is there a way that an attacker could take advantage of this and reduce security? Think about the special case where you only want to encrypt one of two messages: "0" or "1".

Submission instructions

Submit your files using the glookup system on a Hive machine. Enter your source code directory and run the command:

```
submit proj1
```

The files you should submit are: **main.c**, **rsa.c**, and **answers.txt**. Only one member per group needs to submit.

Submissions are due on September 29 at 3:00 PM.