An introduction to Encryption

ft_ssl [rsa] [genrsa]

42 staff staff@42.fr

Summary: This project is a continuation of the previous encryption project. You will code your own random prime number generator and use it to make private RSA keys.

Contents

Chapter I

Foreword

Something Funny Here.

Chapter II

Introduction

The `RSA` algorithm is named after Ron Rivest, Adi Shamir and Leonard Adleman, who invented it in 1977. The RSA cryptosystem is the most widely-used public key cryptography algorithm in the world. It can be used to encrypt a message without the need to exchange a secret key separately. It can be used for both public key encryption and digital signatures.

Its security is based on the difficulty of factoring large integers that are the product of two large prime numbers. (Multiplying these two numbers is easy, but determining the original prime numbers from the total is considered infeasable due to the time it would take even using today's supercomputers.

The public and the private-key generation algorithm is the most complex part of the whole RSA cryptosystem. Two large prime numbers are generated using the `Rabin-Miller` primality test algorithm. A modulus is calculated by multiplying the two numbers, and is used by both the public and the private keys. Its length, usually expressed in bits, is called the key length. The public key consists of the modulus, a public exponent (normally 65537, as it's a prime number that's not too large). The exponent doesn't have to be secretly selected as the public key is shared with everyone. The private key consists of the modulus and a private exponent, which is calculated using the Extended Euclidian algorithm to find the multiplicative inverse with respect to the totient of the modulus.

`There is a lot of complex math terminology going on in here.`

`google man`

Chapter III

Objectives

For this project you will be adding the private key generation part of the `RSA` cryptosystem into your `ft_ssl` executable. You will gain a deeper understanding of the way the entropy of your Unix system is used for cryptographically-secure random numbers, and how it is used in key generation.

You will also gain experience and knowledge in using keys to encrypt and decrypt files and text. If you are smart you will also research:

- Digital signing with a private key
- Verifying digital signatures with a public key
- Sending encrypted messages over untrusted networks without prior key distribution

By the end of this you should be intimately familiar with the workings of asymmetric key cryptosystems.

Chapter IV

General Instructions

- This project will only be corrected by other human beings. You are therefore free to organize and name your files as you wish, although you need to respect some requirements below.
- The executable file must be named `ft_ssl`.
- You must submit a Makefile. The Makefile must contain the usual rules and compile the project as necessary.
- Your project must be written in accordance with the Norm.
- You have to handle errors carefully. In no way can your program quit unexpectedly (Segfault, bus error, double free, etc). If you are unsure, handle the errors like `OpenSSL`.
- You'll have to submit an author file at the root of your repository. You know the drill.
- You are allowed the following functions:
    - `open`
    - `close`
    - `read`
    - `write`
    - `malloc`
    - `free`
- You are allowed to use other functions as long as their use is justified. (Although they should not be necessary, if you find you need `strerror` or `exit`, that is okay, though `printf` because you are lazy is not)
- You can ask your questions on slack in the channel `#ft_ssl`

Chapter V

Mandatory Part

You will be adding additional tools to the `ft_ssl` program you have created.

```
 > ft_ssl
usage: ft_ssl command [command opts] [command args]
```

For this project, you will be adding key generation functionality, using primality number tests and the `RSA` key-generation algorithm and utilities.

`man openssl`

In total you will need to add the commands `genrsa`, `rsa`, and `rsautl`. You will have to code a primality test function to get these to work.

V.0.1 Optimus Prime

For this first part, you must create a function that takes an `unsigned 64 bit long` number and a probability between 0.0 and 1.0 (or 0 to 100, as you see fit) that the given number is prime.

If you don't know what prime numbers are, you should probably check on wikipedia.

See Solovay-Strassen and/or Miller-Rabin algorithms.

```c
/* ************************************************************************** */
/*                                                                            */
/*                                                        :::      ::::::::   */
/*   main.c                                             :+:      :+:    :+:   */
/*                                                    +:+ +:+         +:+     */
/*   By: irhett <irhett@student.42.us.org>          +#+  +:+       +#+        */
/*                                                +#+#+#+#+#+   +#+           */
/*   Created: 1970/01/01 00:01:01 by kwame             #+#    #+#             */
/*   Updated: 2070/02/03 05:08:13 by irhett            ###   ########.fr       */
/*                                                                            */
/* ************************************************************************** */

#include <stdio.h> // <- Don't do this!

int		ft_ssl_is_primary(__uint64_t number, float probability);

int		main(void)
{
	__uint64_t	num;
	float		prob;

	num = 7;
	prob = 0.99;
	if (ft_ssl_is_primary(num, prob))
		printf("%i is prime at %f%% probability.\n", num, prob);
	else
		printf("%i is composite.\n", num);
	return (0);
}
```

You will use this function later to test the primality of the random integers you use later for key generation, so be mindful of your work. If your code tests all cases (fully factors a given integer of that size) it will be too slow, so find a nice balance between accuracy and speed.

By Odin, by Thor! Use your brain!

V.0.2 Ratchet Sideswipe Arcee

The first command that you must add to your `ft_ssl` is `genrsa`. It must generate a private key to the standard output or to a file as specified.

You must implement the following flags: `-rand`, `-i`, and `-o`.

```
> ft_ssl genrsa
Generating RSA private key, 64 bit long modulus
.............................++++++++++++
.....................++++++++++++
e is 65537 (0x10001)
-----BEGIN RSA PRIVATE KEY-----
MIIBOwIBAAJBAMLh8BxMEm/x+wDjpcMAeCANVFUfKdp9XR2H4VAnCK7b3x6SBD0v
lPsWBwcMqRzKYcX6wIOxuDjkdRlMHWyBwb8CAwEAAQJBAKNxQiM5Wa0xUMXqJrdo
yVZ4V2YcgMmRomqF7119nzamEQsDtTeSmA6DP2qbxkYsw22W307jeiOKn5U7031Y
WYECIQDuD/pvMokXEHmPOm4qBX41unlszCD6dZ/IOhxgr1ggkQIhANGRDqEUhnyR
5Yb+N4p3D+n4fdTWBi6voNgFO4IfTuVPAiEAiwteFHCJzaTbuyI/kd+fdbYyka8w
W9kzt/jo9jez22ECIFCbcv0SYAhaNecOsV5ZHY3pPr029XnPpBZzLMiIMliRAiAm
q/e5iyp+zPDMiG2A263x6eQCRbUOXMpU1txEWgCk4w==
-----END RSA PRIVATE KEY-----
```

man genrsa

During moulinette and peer corrections we will only test 64 bit keys, but you should be able to generate keys of any size.

You may not use rand() or time() for your RNG/PRNG!

man urandom

V.0.3 Rollout Smoothly, Autobots

You must also add `ft_ssl rsa` and `ft_ssl rsautl` that behave exactly like `openssl`.

`ft_ssl rsa` must include the following options:

```
ft_ssl rsa [-inform PEM] [-outform PEM] [-in file] [-passin arg] [-out file] [-passout arg] [-des] [-
    text] [-noout] [-modulus] [-check] [-pubin] [-pubout]
```

`man rsa`

`ft_ssl rsautl` must include specifically `encrypt` and `decrypt`, and we recommend that you implement all of these flags for a complete understanding.

```
ft_ssl rsautl [-in file] [-out file] [-inkey file] [-pubin] [-encrypt] [-decrypt] [-hexdump]
```

`man rsautl`

Chapter VI

Bonus part

VI.0.1 DES-epticons

You may also implement the `gendsa` key generation option, and also an option to generate `DES` keys (why don't we call it `gendes`?).

> DSA keys behave like RSA keys in that they are asymmetric public/private as well, however beyond that they are very different. DSA keys are faster at signing documents and generate a smaller signature, but slower at verifying, and can't be used for encryption/decryption.

> man dsa

> man gendsa

VI.0.2 C-C-C-Combo Breaker!

Up until now we have been working with a key size of 64 bits, which is generally considered insecure. Now you get to learn how and why! Come up with any way to break a message encrypted by a 64 bit key.

- Lurk around the Mersenne Twister (oooh~!)
- Brute force break it (booo~!)
- ???
- Profit.

If you attempt this challenge, be nice to your corrector and provide a separate executable with its own Makefile, or include a reasonably named flag or command to your `ft_ssl`.

For example:

```
 > ./ft_breakit
usage: ft_breakit [-k keysize] [-a algo] [-p plaintext] ciphertext_file
 >
 > ./ft_breakit sample.txt
No plaintext provided.
Begining analysis:...
Estimated keysize is 64 bits.
Attempting DES decryption.
<Saving DES log to "sample.txt.des">
...
```

```
 > ./ft_ssl extractkey
usage: extractkey [-k keysize] [-a algo] [-p plaintext] ciphertext_file
```

The executable names, flags and output are suggestions for inspiration, they do not need to be exactly duplicated. You do still need to prove to your corrector that it works.

As you should expect by now, the bonus will not be considered unless the mandatory part is complete and perfect.

Chapter VII

Turn-in and peer-evaluation

Submit your code to your `Git` repository as usual. Only the work in the repository will be considered for the evaluation. Any extraneous files will count against you unless justified.