RSA Introduction and Instructions

Patrick Bradley

11/27/16

What is *simple_rsa*?

This is *simple_rsa*: my own Python 2.7 implementation of a PGP-lite RSA encryption cryptosystem. *simple_rsa* is essentially a much simpler version of GPG, designed to help the novice user get started using the RSA algorithm.

One should use this if they do not want to go through the hassle of using the much more complex GPG cryptosystem, or if they want to see or modify the actual code behind the RSA algorithm, since this code is open-source. Once the concepts explored in this program are understood, additional features are requested, or a more secure cryptograpy is required, the user can graduate to GPG or another PGP-like encryption cryptosystem.

If you're not familiar with asymmetric key encryption, I describe why you would want to use it in the next section. If you're not familiar with the RSA algorithm itself, I describe the algorithm below in the "How does RSA work" subsection of this write-up. Lastly, I describe the syntax employed by *simple_rsa* in the last section: "How do you use *simple_rsa*?"

Why would you use the RSA algorithm?

The RSA algorithm is an *asymmetric* key encryption algorithm. Most ciphers you might be familiar with are so-called *symmetric* key ciphers. If you know the key, you can encrypt your message using the shared symmetric key, and then the person who you have sent the message to decrypts the message using that same shared key. Unless someone knows the key, they will be unable to decrypt and read the message. This kind of cipher has a long and storied history, but it has one significant problem. Let's say you want to communicate secretly with your accomplice, but you haven't already decided on a shared cipher. How do you communicate the shared cipher secretly, so you can begin discussing your conspiracy? This is the problem that the RSA algorithm solves.

The RSA algorithm, as it is *asymmetric*, does not require both parties to have prior knowledge of a shared key. Instead, both parties have two keys associated with them: a public key that they share with anyone else, and a private key which they keep secret. To encrypt a message to your accomplice, you just encrypt the message using his or her public key, which should be readily available to anyone who wants it. The public key does not need to be kept secret. Then, if the accomplice wants to read the message he or she has been sent, he or she decrypts it using his or her own private key. The private key is the ONLY way to decrypt the messages encrypted with the public key—this is what makes the RSA algorithm *asymmetric*, since the public key cannot decrypt content encrypted by the same key. As long as your accomplice has kept his or her own private key safe, he or she will be the only one who is able to

decrypt your message. Using this method, no prior knowledge must be exchanged before sending a secure message. The RSA algorithm allows anyone to encrypt messages to anyone else, even if they are not confidantes. They could be total strangers, so long as they've posted their public key somewhere public. This makes encryption as easy and available as sending an email.

simple_rsa is a Python program designed to help the user get started using the RSA cryptosystem. If you've ever wanted to send an encrypted message to someone, but have not met in person to share a key or cipher, the RSA cryptosystem is for you.

However, a word of warning before we proceed. *simple_rsa* should not be used for real cryptographic purposes. Although I have implemented the RSA algorithm faithfully, thoroughly tested my implementation of this algorithm, and have found no flaws, that does not mean there truly are no flaws in my code. If cryptographic security is or becomes a true priority, consider using a professional encryption algorithm. For this reason, the keys and encrypted messages generated by this program should not interface with actual PGP programs. This is just a school project —a toy—don't trust it with your secrets!

How does RSA work?

The RSA algorithm was developed by three MIT mathematicians in 1977. Rivest, Shamir and Adleman spent nearly a year racking their brains to come up with a one-way function that would be difficult to invert, before finally discovering that factoring large prime numbers met this requirement. Obviously, the math behind the RSA algorithm can be difficult to understand if you are not a mathematician, but I will attempt to simplify the process the RSA algorithm goes through to encrypt and decrypt messages to the level of the average user with some math background.

Generating a Key

Before using RSA to encrypt or decrypt messages, a public key and a private key must be generated. These keys are bound to one another—they are two halves of the same whole. A message encrypted with one can only be decrypted by the other, and vice versa. As RSA is *asymmetric*, a message encrypted with one half will not be able to be decrypted with that same half.

The basic principle of RSA is that given three very large integers, *e*, *d*, and *n*, for:

$$(m^e)^d \equiv m \bmod n$$

Even knowing *e* and *n* (which together form the public key), it is extremely difficult to find *d* (the private key), where *m* is the message.

To generate your public and private key:
(1) First generate two large primes, *p* and *q*.
(2) Multiply *p* and *q* together. The product is *n*. *n* is the modulus for both the public and private keys. Therefore it's length is the key length (typically 1024 – 4096 bits).

The key concept behind the RSA algorithm is that if you know *p* and *q*, generating *n* is extremely simple. All you have to do is multiply. However, given a large enough *n*, it is nearly impossible to factor it back it into its two factors, *p* and *q*. For instance, let's say you have an *n* = 3763 (obviously in real usage this would be much larger). Can you find the two factors of *n* easily? But if you know that *p* and are 53 and 71, then *n* is easy to find.

(3) Compute *tot(n) = n - (p + q – 1)*. *tot(n)* is the totient of n.
(4) Choose an integer *e* that is coprime to *tot(n)*. Their only common divisor should be 1. This is done by randomly testing numbers within the range $2^{keysize-1}$ through $2^{keysize}$. This integer *e* and the modulus *n* is the public key.
(5) Choose an integer *d* that is the modular multiplicative inverse of *e mod tot(n)*. The product of *d* and *e* modulus *tot(n)* should be 1. For instance, if *e* is 3 and *n* is 11, then you are trying to find *d*, such that 3*d = 1 within the ring of integers mod 11. In this case, *d* = 4, since 3*4 = 12, and 12 mod 11 = 1.

Now you have your public key, which is *e* and *n*, and your private key, which is *d* and *n*.

Send your public key out into the ether, but keep your private key secret.

Encrypting a message

Say you wish to encrypt a message to Bob. Given Bob's public key, take your message and convert it into an integer. For instance, if your message is ASCII, convert it into its byte representation.
Then raise your message to the power of Bob's public key's *e*, and then modulus it by Bob's public key's *n*. Modular exponentiation makes this process extremely efficient.

c = m^e mod n, where *c* is the encrypted message, also known as the ciphertext.

Then send c to Bob.

Decrypting a message

Bob receives the message *c* from you. To decrypt it, he takes his private key *d*, and does modular exponentiation on the ciphertext.

m = c^d mod n, where m should be the original, decrypted message.

Now Bob has the original message. No one else, even if your ciphertext was intercepted, can decipher it.

How do you use *simple_rsa*?

simple_rsa implements the algorithm above, and gives the user access to several commands through which to interface with the algorithm. In this section, I will give examples of the process the user might go through, using *simple_rsa's* syntax.

Choose a folder to keep this program in. The important files and directories that are required to run *simple_rsa* are main.py and the accompanying RSA folder.

You will also need Python 2.7 (https://www.python.org/downloads/release/python-2712/) installed on your machine.

To run *simple_rsa*, open a command-line terminal, and navigate to the folder containing main.py. Then, type:

```
python main.py
```

This should display all the options available to you.

The first step to using this is to generate your public and private keys. Type:

```
python main.py gen_key
```

Give it a couple seconds and simple_rsa should output two *.asc files, one containing your public key and one containing your private key. Obviously, keep your private key safe and secret if you plan on using it.

It should also generate a *keyring.p* file. This contains information about which keys you have access to. Think about it like a phone book, associating names to the location of the public key on your computer. This is so you don't have to keep track of the location of all the public key's you've accumulated yourself.

Say you now want to send a message to Bob. Bob sends you his public key. I've attached a file called Bob_Example_Public_Key.asc with the rest of this program. To import this key into your keyring, type:

```
python main.py import Bob Bob_Example_Public_Key.asc
```

The first argument after import is the name you want to associate the public key with, and the second argument is the location of the public key itself. Now, type:

```
python main.py list_keys
```

Bob's public key should be displayed alongside your self_public and self_private key in the list that is displayed.

Now, create the message you want to send to Bob. Create a *.txt file and type your message. It can be any ASCII message. Save the *.txt file. Type:

```
python main.py encrypt [name of *.txt file] Bob
```

For instance, I typed `python main.py encrypt secret_message.txt Bob.`

The message should be encrypted and a file called [name of *.txt file].pgp should be generated by the script. Bob could decrypt this using his own private key. But obviously you cannot since you don't have the his private key. Now we'll test decryption. Encrypt the same message using your own public key. Type:

```
python main.py encrypt [name of *.txt file] self_public
```

‘self_public’ is the name of your public key. Now, to decrypt using your private key, type:

```
python main.py decrypt [name of *.txt file].pgp
```

This should output a decrypted version of the text file.

And that's it. Now you can use the RSA cryptosystem yourself to generate keys, encrypt and decrypt messages using state-of-the-art encryption.