**RSA Algorithm Example**

* Choose p = 3 and q = 11
* Compute n = p \* q = 3 \* 11 = 33
* Compute φ(n) = (p - 1) \* (q - 1) = 2 \* 10 = 20
* Choose e such that 1 < e < φ(n) and e and φ (n) are coprime. Let e = 7
* Compute a value for d such that (d \* e) % φ(n) = 1. One solution is d = 3 [(3 \* 7) % 20 = 1]
* Public key is (e, n) => (7, 33)
* Private key is (d, n) => (3, 33)
* The encryption of *m = 2* is *c = 27 % 33 = 29*
* The decryption of *c = 29* is *m = 293 % 33 = 2*

The RSA algorithm is the most widely used [Asymmetric Encryption](https://www.practicalnetworking.net/series/cryptography/asymmetric-encryption/) algorithm deployed to date.

The acronym is derived from the last names of the three mathematicians who created it in 1977:  Ron **R**ivest, Adi **S**hamir, Leonard **A**dleman.

In order to understand the algorithm, there are a few terms we have to define:

* **Prime**– A number is said to be Prime if it is only divisible by 1 and itself.  Such as: 2, 3, 5, 7, 11, 13, etc.
* **Factor** – A factor is a number you can multiple to get another number.  For example, the factors of 12 are 1, 2, 3, 4, 6, and 12.
* **Semi-Prime** – A number is Semi Prime if its only factors are prime (excluding 1 and itself). For example:
**12** is *not* semi-prime — one of its factors is 6, which is not prime.
**21** *is* semi-prime — the factors of 21 are 1, **3**, **7**, 21.  If we exclude 1 and 21, we are left with 3 and 7, both of which are Prime.
*(Hint: Anytime you multiply two Prime numbers, the result is always Semi Prime)*
* **Modulos** – This is a fancy way of simply asking for a [remainder](https://en.wikipedia.org/wiki/Remainder).  If presented with the problem 12 MOD 5, we simply are asking for the remainder when dividing 12 by 5, which results in 2.

With that out of the way, we can get into the algorithm itself.

**RSA Key Generation**

The heart of Asymmetric Encryption lies in finding two mathematically linked values which can serve as our Public and Private keys.  As such, the bulk of the work lies in the generation of such keys.

To acquire such keys, there are five steps:

1. **Select two Prime Numbers:  P and Q**

This really is as easy as it sounds.  Select two prime numbers to begin the key generation.  For the purpose of our example, we will use the numbers **7** and **19**, and we will refer to them as **P** and **Q**.

1. **Calculate the Product: (P\*Q)**

We then simply multiply our two prime numbers together to calculate the product:

7 \* 19 = **133**

We will refer to this number as **N**.  Bonus question: given the terminology we reviewed above, what kind of number is N?

1. **Calculate the Totient of N: (P-1)\*(Q-1)**

There is a lot of clever math that goes into both defining and acquiring a [Totient](https://en.wikipedia.org/wiki/Euler%27s_totient_function%22%20%5Ct%20%22_blank).  Most of which will be beyond the intended scope of this article.  So for now, we will simply accept that the formula to attain the Totient on a Semi Prime number is to calculate the product of one subtracted from each of its two prime factors. Or more simply stated, to calculate the Totient of a Semi-Prime number, calculate P-1 times Q-1.

Applied to our example, we would calculate:

(7-1)\*(19-1) = 6 \* 18 = **108**

We will refer to this as **T** moving forward.

1. **Select a Public Key**

The Public Key is a value which must match three requirements:

* It must be Prime
* It must be less than the Totient
* It must NOT be a factor of the Totient

Let us see if we can get by with the number 3:  3 is indeed Prime, 3 is indeed less than 108, but regrettably 3 is a factor of 108, so we can not use it.  Can you find another number that would work?  Here is a hint, there are multiple values that would satisfy all three requirements.

For the sake of our example, we will select **29** as our **Public Key**, and we will refer to it as **E** going forward.

1. **Select a Private Key**

Finally, with what we have calculated so far, we can select our Private Key (which we will call **D**).The Private Key only has to match one requirement:  The Product of the Public Key and the Private Key when divided by the Totient, must result in a remainder of 1.  Or, to put it simply, the following formula must be true:

(D\*E) MOD T = 1

There are a few values that would work for the Private Key as well.  But again, for the sake of our example, we will select **41**.  To test it against our formula, we could calculate:

(41\*29) MOD 108

We can use a calculator to validate the [result is indeed 1](https://www.google.com/search?q=(29*41)+mod+108). Which means **41** will work as our Private Key.

And there you have it, we walked through each of these five steps and ended up with the following values:



Now we simply pick a value to be used as our Plaintext message, and we can see if Asymmetric encryption really works the way they say it does.

For our example, we will go ahead and use **99**as our Plaintext message.

(the math gets pretty large at this point, if you are attempting to follow along, I suggest to use the [*Linux Bash Calculator*](http://www.folkstalk.com/2012/09/bc-command-examples-in-unix-linux.html) utility)

### Message Encryption

Using the keys we generated in the example above, we run through the Encryption process.  Recall, that with Asymmetric Encryption, we are [encrypting with the Public Key, and decrypting with the Private Key](https://www.practicalnetworking.net/series/cryptography/using-asymmetric-keys#asym-encryption).

The formula to Encrypt with RSA keys is:  **C**ipher Text = **M^E MOD N**

If we plug that into a calculator, we get:

99^29 MOD 133 = **92**

The result of **92** is our Cipher Text.  This is the value that would get sent across the wire, which only the owner of the correlating Private Key would be able to decrypt and extract the original message.  Our key pair was 29 (public) and 41 (private).  So lets see if we really can extract the original message, using our Private Key:

The formula to Decrypt with RSA keys is:  Original **M**essage = **M^D MOD N**

If we plug that into a calculator, we get:

92^41 MOD 133 = **99**

As an experiment, go ahead and try plugging in the Public Key (29) into the Decryption formula and see if that gets you anything useful.  You’ll notice that, as was stated before, it is impossible to use the same key to both encrypt and decrypt.

### Message Signing

But remember, [that isn’t all we can do with a key pair](https://www.practicalnetworking.net/series/cryptography/using-asymmetric-keys#asym-signing).  We can also use same key pair in the opposite order in order to verify a message’s signature.

To do this, we will use the same formulas as above, except this time we will revere the use of the Public and Private Key.  We’re going to encrypt with the Private Key and see if we can decrypt with the Public Key.

We’ll use the same formula to encrypt, except this time we will use the Private Key:

**S**ignature = **M^D MOD N**

If we plug that into a calculator, we get:

99^41 MOD 133 = **36**

The result of **36** is the Signature of the message.  If we can use the correlating public key to decrypt this and extract the original message, then we know that only whoever had the original Private Key could have generated a signature of 36.

Again, the same Decryption formula, except this time we will use the Public Key:

Original **M**essage = **M^E MOD N**

If we plug that into a calculator, we get:

36^29 MOD 133 = **99**

And there you have it.  A fully working example of RSA’s Key generation, Encryption, and Signing capabilities.

It should be noted here that what you see above is what is regarded as “vanilla” RSA.  In production use of RSA encryption the numbers used are significantly larger. In fact, modern RSA best practice is to use a key size of 2048 bits.  This correlates to the ***N*** value in our calculation above.  The two primes used in modern RSA must result in a product that is 2048 bits.

And just to give you an idea of how big 2048 bit number is. We saw earlier that a 128 bit number can be written in decimal with [39 digits](https://www.practicalnetworking.net/series/cryptography/symmetric-encryption#key-sizes). A 2048 bit key is exponentially longer – it would require approximately 617 digits to fully write out.

Problems: [RSA Encryption Algorithm - Javatpoint](https://www.javatpoint.com/rsa-encryption-algorithm)

[Elliptic Curve Cryptography (ECC) - Practical Cryptography for Developers (nakov.com)](https://cryptobook.nakov.com/asymmetric-key-ciphers/elliptic-curve-cryptography-ecc)

[RSA Calculator (drexel.edu)](https://www.cs.drexel.edu/~popyack/IntroCS/HW/RSAWorksheet.html)