Deepening : Encrypting Messages with RSA

Jon Camfield

2024-0w

# Materials

* Work through this yourself in advance to make sure you’re comfortable with the process.
* Whiteboard or flipchart
* Participants will need calculators - the simple calculator app on phones or laptops is sufficient.

This lesson first goes through the encryption and decryption process with pre-selected numbers, and then follows with a short explanation of how you can independently create the “right” numbers that work in this situation.

*The author would like to thank Michael Starbird for inspiring this lesson and for his walk through of RSA in* [*Coincidences, Chaos, and All that Math Jazz*](https://www.google.com/books/edition/Coincidences_Chaos_and_All_that_Math_Jaz/x-uQ2aiDveUC?hl=en&gbpv=1&dq=michael+starbird+secrets&pg=PA65&printsec=frontcover)*.*

# Deepening

We are going to actually encode and messages, and learn a few bits of the math behind how public/private key cryptography actually works.

*This lesson builds off of “Secrets Held, Secrets Revealed” in* [*Coincidences, Chaos, and all that Math Jazz*](https://www.google.com/books/edition/Coincidences_Chaos_and_All_that_Math_Jaz/x-uQ2aiDveUC?hl=en&gbpv=1&dq=michael+starbird+secrets&pg=PA65&printsec=frontcover) *by Edward Burger and Michael Starbird. If you want to diver deeper into this, I highly recommend it.*

The private and public keys in asymmetric cryptography - be that PGP, TLS certificates, or other places you’ll find it - are numbers. Very – very large numbers. And specifically, they are numbers with some very weird inter-connections, based on prime numbers (numbers like 11, but unlike 25, which can only be evenly divided by themselves and the number 1) and a calculation called “modular” arithmetic.

To do this by hand, we are going to use very small numbers. In real life applications, these are numbers with **hundreds** or even **thousands** of digits. (you can explore this with a few tools like pgpdump https://github.com/kazu-yamamoto/pgpdump !)

## RSA By Hand

### Creating Public and Private Keys

To ground this exercise a bit, we have some pre-selected numbers we’ll be using for the example, but these instructions give you the tools to choose and build your own.

p = 11 (first prime) q = 13 (second prime) n = 143 (the modulus, p×q) ϕn = 120 (the totient, (p-1)×(q-1) ) e = 7 (public exponent, another prime that is … d = 103 (the modular inverse of n)

1. First off, we are going to select two small prime numbers. Prime numbers are numbers that are only divisble by themselves and the number one. We will call these two primes **p** and **q**.
2. Multiply p and q together, note down the result. We will call this number **n**, and it will be part of both the public and the private key.
3. With the same two primes p and q, subtract 1 from each prime and multiply together the result. So: (p-1)×(q-1). We will call the result of this **phi n** (or **ϕn**).
4. Select a third prime (we’ll call it **e**), but it has to be one which doesn’t divide evenly into ϕn from the previous step. (so, if the result was 65, 5, while itself a prime, is a bad choice). This third prime will be the other part of your private key and will also be used to compute your public key.
5. This is the hard step! We need to find what we’ll call **d**, which will be the “modular multiplicative inverse” of n.

* The “official” way to find this is called the [Extended Euclidean algorithm](https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm). However, this is when we remind ourselves that we’re not becoming cryptographers today - there is a ton of number theory behind how this works, where it doesn’t, and more. BUT - it is all actual math, with steps one can follow by hand - that underpins all of this.
* For today we’re taking a shortcut, but still involving a core piece of the math used in encryption - **modular arithmetic**. Perhaps you remember when you were first learning division, and had “remainders” - divide 10 by 7 on a calculator, and you’d get 1.428571…. If you calculate remainders, you’d get *1r3* - 7 goes into 10 once, with 3 remaining. With modular arithmetic, this remainder value is what we are trying to find - so 10 mod 7 is 3. 17 mod 7 (10+7) is also 3 - 7 goes into 17 twice (14), and the remainder is still 3. It’s also true that 13 mod 10 is 3. 103 mod 100 is 3 - it’s kinda a funny tool, useful for thinking about 12 vs 24hr clock time (3pm is 15:00 mod 12!) and, in this case, incredibly useful for security.
* ANYHOW - to find the modular multiplicative inverse, we are going to look for an integer - a number with no decimals - that multiplies with e, that ϕn - the number from step 3 - modulo this is 1.
  + Take ϕn, and add 1
  + Divide that by e. If we get a simple integer, that’s our answer. The “add 1” step here is to ensure that we find the mod = 1 result.
  + If we got a number with decimals, we keep going - each step from here multiplies the “step 3 number” incrementally - 2x, 3x, 4x, etc., then adds 1, and then divides by e - until we get a plain integer. With the numbers 7 and 120, here is the process:
* Try 1: (120 + 1)/7 121/7 = 17.2857… Try 2: (120 \* 2 +1) / 7 = 241/7 = 34.428571429 Try 3: (120 \* 3 +1) / 7 = 361/7 = 51.571428571 Try 4: (120 \* 4 +1) / 7 = 481/7 = 68.714285714 Try 5: (120 \* 5 +1) / 7 = 601/7 = 85.857142857 Try 6: (120 \* 6 +1) / 7 = 721/7 = 103 …WOOT!
  + BINGO! The number we’re looking for is 103. This number will also always itself be smaller than the number we are dividing in to, because that would simply cause us to loop over again - so if you kept going, you will find more numbers that work, but it is this first one, which is less than the number you’re dividing in to from Step 3, which is the one that successfully completes the math.

1. Now we (finally!) have all the pieces to build our public and private keys. It’s worth noting that technically and mathematically, either of these keys could be either the public or private key - they are effectively mirrors of each other - which unlocks some super fun tricks we’ll get to soon. Traditionally with RSA, we use e and n for the public key, and d and n for the private key, and this is usually written like this: Public Key: (e,n) , Private Key: (d,n). In fact, you can use pgpdump’s web interface on a sample or public key and see it provide the n and e values of the key! It’s worth noting that in most cases, e is set to 65537 (01 00 01 in hex) as a default.

### Public and Private Keys; Encrypting and Decrypting

encrypted = mod *n* ( (message)^*e* )

decrypted = mod *n* (encrypted ^*d* )

### Example Run-through

Let’s start with a super simple, one-letter message - “y”, and translate that letter into a number - 25.

p,q = 11, 13 (the primes) n = 143 (the modulus) ϕn = 120 (the totient) e = 7 (public exponent) d = modular inverse (n) = 103 (the private exponent)

https://www.google.com/books/edition/Coincidences\_Chaos\_and\_All\_that\_Math\_Jaz/x-uQ2aiDveUC?hl=en&gbpv=1&dq=michael+starbird+secrets&pg=PA65&printsec=frontcover

https://www.geeksforgeeks.org/euclidean-algorithms-basic-and-extended/ https://nhoyle-unsw.github.io/learn-encryption-with-python/Modular-arithmetic.html https://doctrina.org/How-RSA-Works-With-Examples.html https://doctrina.org/Why-RSA-Works-Three-Fundamental-Questions-Answered.html https://en.wikipedia.org/wiki/Extended\_Euclidean\_algorithm#Example https://ccom.uprrp.edu/~humberto/very-small-rsa-example.html