# X.509 Certificates

RSA (Rivest–Shamir–Adleman) was invented in 1979. And if you are Italian, it doesn’t mean ‘Residenza Sanitaria Assistenziale’. The internet wouldn’t be the same without them, since no secure financial transaction would exist. No internet banking, no ssl, no https.

After a problem with VPN certificates, I decided to dive deeper into how they are made and how they work. This is how they look like, opening one of them with a text editor:

**-----BEGIN CERTIFICATE-----**

**MIICYzCCAcygAwIBAgIBADANBgkqhkiG9w0BAQUFADAuMQswCQYDVQQGEwJVUzEM**

**MAoGA1UEChMDSUJNMREwDwYDVQQLEwhMb2NhbCBDQTAeFw05OTEyMjIwNTAwMDBa**

**Fw0wMDEyMjMwNDU5NTlaMC4xCzAJBgNVBAYTAlVTMQwwCgYDVQQKEwNJQk0xETAP**

**BgNVBAsTCExvY2FsIENBMIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQD2bZEo**

**7xGaX2/0GHkrNFZvlxBou9v1Jmt/PDiTMPve8r9FeJAQ0QdvFST/0JPQYD20rH0b**

**imdDLgNdNynmyRoS2S/IInfpmf69iyc2G0TPyRvmHIiOZbdCd+YBHQi1adkj17ND**

**cWj6S14tVurFX73zx0sNoMS79q3tuXKrDsxeuwIDAQABo4GQMIGNMEsGCVUdDwGG**

**+EIBDQQ+EzxHZW5lcmF0ZWQgYnkgdGhlIFNlY3VyZVdheSBTZWN1cml0eSBTZXJ2**

**ZXIgZm9yIE9TLzM5MCAoUkFDRikwDgYDVR0PAQH/BAQDAgAGMA8GA1UdEwEB/wQF**

**MAMBAf8wHQYDVR0OBBYEFJ3+ocRyCTJw067dLSwr/nalx6YMMA0GCSqGSIb3DQEB**

**BQUAA4GBAMaQzt+zaj1GU77yzlr8iiMBXgdQrwsZZWJo5exnAucJAEYQZmOfyLiM**

**D6oYq+ZnfvM0n8G/Y79q8nhwvuxpYOnRSAXFp6xSkrIOeZtJMY1h00LKp/JX3Ng1**

**svZ2agE126JHsQ0bhzN5TKsYfbwfTwfjdWAGy6Vf1nYi/rO+ryMO**

**-----END CERTIFICATE-----**

This is a ‘base64’ representation of a sequence of bits. If we want to deal with digital certificates, we need a common view on how they should be represented, to avoid problems. From Wikipedia we get a table with the character representation for a sequence of 6 bits. Since we usually transmit sequence of bytes, there could be the need for ‘padding’ or adding a sequence of bits to reach a multiple of 8 bits.

But if you save such a certificate on your PC and you open it, you will probably see something like this:

A screenshot of a computer

Description automatically generated

You don’t type-in a password, this means that this ‘.cer’ file can be parsed and understood by anyone who know how it works.

A screenshot of a computer screen

Description automatically generated

After all, a certificate is just a sequence of information. What are the rules ? they are written using ‘**Abstract Syntax Notation’**, which is a standard way to store information (like the more recent YANG). This is the best complete explanation of how it works, using a very good example: Google’s Certificate.

<https://github.com/ajanicij/x509-tutorial/blob/master/x509-analysis.md>

Copying something from the above site, this is how a certificate is written:

**Certificate ::= SEQUENCE {**

**tbsCertificate TBSCertificate,**

**signatureAlgorithm AlgorithmIdentifier,**

**signatureValue BIT STRING**

**}**

What this says is that a Certificate is a SEQUENCE, which is similar to a C struct. That SEQUENCE consists of three parts: **tbsCertificate**, which has type TBSCertificate (I believe this stands for **to-be-signed certificate**), **signatureAlgorithm** of type AlgorithmIdentifier, and **signatureValue** of type BIT STRING.

**DER**

DER stands for Distinguished Encoding Rules and it is the ASN.1 encoding that is used for encoding a certificate. It is a subset of BER, but such that for any ASN.1 data structure there is **exactly one way to encode it in DER**. In DER every ASN.1 element, primitive or composed, is encoded as a TLV triplet, where TLV stands for Tag-Length-Value. Here's a simple example, integer 2:

02 01 02

The first byte 02 is the tag for integer. The second byte, 01, is the length of the value part. The value part contains just one byte, with the value 02.

To be honest, this doesn’t make your life so simple. Even in case you want to write down a Python script that parses a X.509 certificate, it would require quite a lot of work and would be a long way to go. Luckily, there is a Python library named ‘pyasn’ that does it for us. You can find a few examples on the internet, and it’s not so straightforward anyway. For example, there are a few references to RFC because probably some minor details have been changed in the way some specific things are managed. Depending on the decoding called function, you can access the certificate’s data as a dictionary or calling some other functions.

**from pyasn1.codec.der.decoder import decode**

**from pyasn1.codec.der.encoder import encode**

**from pyasn1\_modules import pem, rfc2459**

**from hashlib import sha1, sha256**

**import pyasn1\_modules.rfc5280**

**import pyasn1\_modules.rfc2437**

**import pyasn1\_modules.rfc2315**

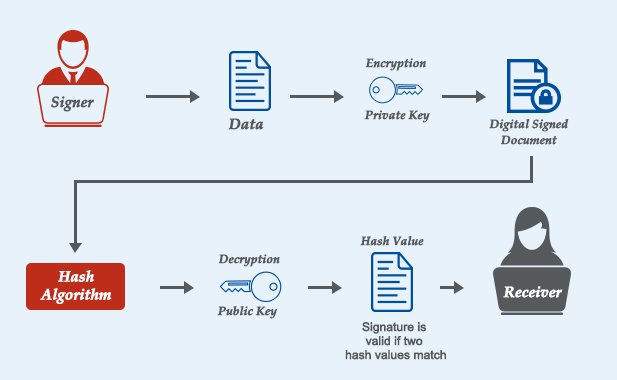
But in the end, **what did we want to do ?** We wanted to write a Python script that receives 2 certificates in input, and **checks if the first certificate has been signed by the second one**, which is the ‘Certificate Authority’. In case they are the same certificate, you will check if it is a self-signed certificate. Moreover, a couple of libraries like ‘Crypto’ or ‘cryptography’ are already written to do the full job for you, but one of them has been ported to Rust and was not available in my PC.

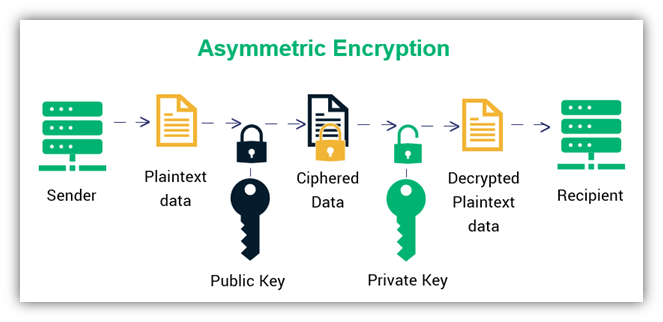
<https://cryptography.io/en/latest/faq/#why-does-cryptography-require-rust>

So I decided to **write a less ‘high level’ script**, which of course requires a deeper understanding of how things work.

# Digital Signatures

The big revolution of **RSA**, was the ability to encrypt and decrypt messages WITHOUT sharing a common password (which would be an ‘unsecure’ or impossible way to manage the transmission of encrypted packets). Such an approach is quite CPU intensive and doesn’t scale globally on a network like the internet, so what is usually done, is using asymmetric cryptography to transmit securely a shared common password, and from this point on, symmetric encryption will be used. This is done by encrypting data with the peer’s public key: only having the private key, it is possible to decrypt the message.





But another way to use RSA is that of digitally SIGNING data. What does this mean ? This means that given a defined PUBLIC key that is known to everyone, you can produce using the private key a sequence of bytes for a certain document, that can be used to verify the signature by anyone, ideally in a very simple and fast way. If the check is fine, this means that only the identity who knows the private key associated to the public one, is the one who has signed the document.

Beware that RSA was invented in 1979 (a long way ago), when the internet didn’t really exist yet. Later other encryption algorithms like ECDSA (Elliptic Curve Digital Signature Algorithm) have been developed, but RSA is still widely used. ECDSA is presently used in Bitcoin blockchain to sign transactions. The advantage of ECDSA is that it is more efficient respect to RSA: given a certain common key length, ECDSA is ‘stronger’ to break than RSA. Comparisons between the two can occupy weeks or months of studies.

# RSA, the math behind it

Thanks to ‘**Ayoub Omari**’ for the best and most simple (for how simple it can be …) explanation about the math behind RSA and how it works:

<https://towardsdatascience.com/the-math-behind-rsa-910f88b94c36>

Of course we can’t verify a digital signature made with RSA unless we understand how the algorithm works. Understanding the math behind it adds some magic to it, but it’s not really needed. Copying from the above link, the public and private keys we need are generated in the following way:

1. Randomly generate two large prime numbers *p* and *q* of size 2048 bits each
2. Compute *N = p•q* and *φ(N) = (p-1)•(q-1)*
3. Choose a number ***e*** coprime with *φ(N)*
4. Using [Euclid extended algorithm](https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm), compute ***d*** the inverse of ***e*** modulo *φ(N)*
5. Store *(e, N)* as the public key, and *(d, p, q, N)* as the private key.

Let’s now suppose we have Bob requesting to Alice to ‘sign’ a book digitally. Bob sends data to Alice, and here comes the first problem: how can we sign something so big ? the problem is solved by using hash algorithms. There’s a lot of math and theory behind these algorithms, but what they do is producing a sequence of bytes of a fixed, deterministic length. If you change one bit on the input message, the output is going to be completely different. SHA1 has been deprecated as a hashing algorithm, so nowadays SHA-2 is used (with an output length of 224, 256, 384 or 512 bits). Such value can be ‘padded’ to be as long as the private/public key length (for example 2048 bits).

So once you have the hashed message ‘m’ (padded if required), Alice can now calculate the signature using its private key ‘d’:

s = (mod *N*)

How do we verify that such signature is correct and has been produced by Alice ? how do we ensure that Alice MUST know the private key ‘d’, without us (being the verifiers) knowing it ? we need to verify if:

(mod *N*) == m

The above equation, if the signature is correctly calculated, is true for every ‘m’. There’s some MAGIC in maths … especially if you look at real life examples and numbers in the next paragraph. And this is where you could be interested in studying the math behind RSA (starting from the above link).

# Python script

Asking to ChatGPT didn’t help so much to produce such a script, telling him to not use libraries I couldn’t use because they are written in Rust (I don’t have Rust installed in my windows PC). To put on the table all the things we need to know, there is also the ‘padding’ algorithm to be used with the hashing output. A ‘simple’ one is ‘**pkcs1.5’**, that basically works like this:

[**https://medium.com/@bn121rajesh/rsa-sign-and-verify-using-openssl-behind-the-scene-bf3cac0aade2**](https://medium.com/@bn121rajesh/rsa-sign-and-verify-using-openssl-behind-the-scene-bf3cac0aade2)

**0x00 || 0x01 || (0xFF \* padding) || 0x00 || ‘digestInfo’ || HASH\_OUTPUT**

'DigestInfo' is a fixed and well known sequence of bytes that depends on the used hashing algorithm, for **SHA-1** (the deprecated algorithm) is the following hex value:

**0x3021300906052b0e03021a05000414**

For **SHA-256** ( or SHA-2 with 256 bits) it’s the following:

**0x003031300d060960864801650304020105000420**

How do you know which cryptographic algorithm has been used to sign the certificate, and which padding has been used ? It is of course something VERY important. The snmp-like object ID that represents the signature algorithm (included in the certificate), also states which padding algorithm is used, so there can be no doubts about it.

<https://cryptography.io/en/latest/x509/reference/>

Just a couple of examples:

**RSAES\_PKCS1\_v1\_5**

Corresponds to the dotted string "1.2.840.113549.1.1.1". This is a RSAPublicKey public key with PKCS1v15 padding.

**RSASSA\_PSS**

Corresponds to the dotted string "1.2.840.113549.1.1.10". This is a RSAPublicKey public key with PSS padding (not covered in this script).

The script can be also found in my Github page:

<https://github.com/ricky-andre/Crypto/blob/master/X509_cert_signature_verify.py>

Here we just comment and highlight the most important steps, also providing a real life example.

The most important function is of course ‘verify\_signature’, passed parameters are the certificate to be verified and the CA (‘Certification Authority’) certificate, containing the public key and the public exponent. In most cases, public exponent will be , which requires even with big numbers only 16 steps to be computed:

🡪 🡪 …. 🡪 🡪

The script only works with RSA and with SHA-1 or SHA-2 with 256 bits.

**def verify\_signature (cert, ca\_cert):**

**signature = cert["signatureValue"]**

**tbs\_cert = encode(cert["tbsCertificate"])**

**sign\_algo = (str)(cert["signatureAlgorithm"]["algorithm"])**

**# we retrieve from the CA the public key and the public exponent**

**ca\_mod, ca\_exp = public\_key\_from\_certificate (ca\_cert)**

**# now we calculate s^e mod N**

**signed\_int = pow(int(signature), ca\_exp, ca\_mod)**

**signed\_bytes = signed\_int.to\_bytes(len(signature) // 8, "big")**

**# sha1, rsa1 with pkcs 1.5 padding**

**if (sign\_algo == "1.2.840.113549.1.1.5"):**

**hash\_bytes = sha1(tbs\_cert).digest()**

**if not re.search("^0001[f]+003021300906052b0e03021a05000414", str(signed\_bytes.hex())):**

**return False**

**else:**

**# check the last 160bits of the signature, if they are the same it's ok**

**return hash\_bytes == signed\_bytes[-20:]**

**# sha256, rsa1 with pkcs 1.5 padding**

**elif (sign\_algo == "1.2.840.113549.1.1.11"):**

**hash\_bytes = sha256(tbs\_cert).digest()**

**if not re.search("^0001[f]+003031300d060960864801650304020105000420", str(signed\_bytes.hex())):**

**return False**

**else:**

**# check the last 256bits of the signature, if they are the same it's ok**

**return hash\_bytes == signed\_bytes[-32:]**

**return False**

Let’s provide an example of the numbers we’re dealing with, we’re using a CA with a key length of **2048** bits, the message goes through SHA256 with pkcs1.5 padding. How many digits do we need in base 10 to represent a 2048 bit long number ?

The script has been proven to work even with **8192** bits long keys, which means decimal numbers of about **2466** digits. We need to thank Python here, because it has embedded support for virtually infinite integer numbers, the only limit being the PC memory to manage them.

**Signature** ‘s’ example from a real certificate, decimal number (617 digits):

**17053997448177446999993647732686366185508333595048346998466655339470783939788969560524929728926666854825407644877936723344338890208168289141114524333530486078353150718832863029588654642742684705843033703768245818850876527810168343017458997127420558017292060115147526462830777916358005432394070769673549586876735989066408257519026385910556548563004716711813694602873776460994272249017835217016437203004647131398204765582525226931725220538372746093864946440481567454266211098215172062603180752516780193172852244736499434936574806489144965758903568321164107447620687356443435928552577940420344414735847011852771641833607**

*N* value, operations are performed modulus N (617 digits):

**26791771534313539947178574731723256559937856847353259186492505026941841971436519674778928107669441854001401438075211695487607448674303061753607748679895966484271924851904393530005707096259844215860044641551589264765030431818084802738313697326691585946106458399440905929126992959615105959579819994115400995950629324817754977527738118432162799246775255332992686539840259783553828653623693138376324694600319266327532569660109661577980239974582147411923150655914246120367282613537060213815940977405957083585581391497772469144753471471195330129154394948054557262861402147515549335337581984246148617811987010227115476589923**

(mod *N*), the supposed ‘m’ value:

**986236757547332986472011617696226561292849812918563355472727826767720188564083584387121625107510786855734801053524719833194566624465665316622563244215340671405971599343902468620306327831715457360719532421388780770165778156818229863337344187575566725786793391480600129482653072861971002459947277805295727097226389568776499707662505334062639449916265137796823793276300221537201727072401742985542559596685092673521228140822200236743113743661549252453726123450722876929538747702356573783116197523966334991563351853851212597377279504828784709728051249662286782637609097112763086177541888131952885428908055184613891586**

‘m’ value in hex:

****

If we calculate the hash on the ‘**To Be Signed Certificate’**, we obtain the following hex value:

**sha256(tbs\_cert).digest() = 0x5702b6918ef882d88ed5f83baf8545f8f8f0e9eddf55899e2021316105acde02**

It looks like that the certificate has been signed by the certificate Authority. Is there an easy way to produce a ‘fake’ signature ? you should guess the private key that is used to produce the signature from the (hashed) message. Given we’re working with 2048bits keys, or numbers with 617 digits, this is virtually impossible, whatever is your available computational power.

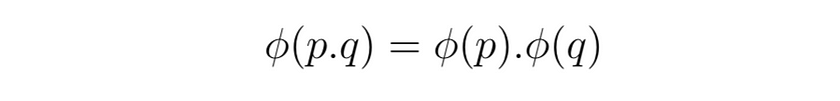
# Prime numbers and modular math

Follows hereafter the copied and pasted article about RSA math written by ‘**Ayoub Omari**’. It can’t be shorter than this, but there’s everything you need: pure elegance.

**Euler’s totient function**

Consider ***φ(N)*** the number of strictly positive numbers **less than** *N* and **relatively prime** with *N*.

For example *φ(8) = 4*, because there are 4 integers less than and coprime with 8 which are 1, 3, 5, and 7. It can be shown that for any two **coprime** integers *p* and *q*:



Think about it. By knowing how many integers are coprime and less than *p,* and how many are coprime and less than *q*, we know how many integers are coprime and less than *N=p.q* without ever dealing with the integers between *p* and *p.q* and between *q* and *p.q*.

An example may help highlight this magic equation.

Take the number 72. It is equal to 8x9 which are coprime.

* *φ*(8) = 4 : Integers coprime with 8 are 1, 3, 5, 7
* *φ*(9) = 6 : Integers coprime with 9 are 1, 2, 4, 5, 7, 8

The equality above tells us that there are 4x6=24 numbers coprime and less than 72. We didn’t even think about these numbers and how they relate to 72, but we know how many there are that are coprime and less than 72. And this is some of the magic of mathematics.

**Euler’s theorem**

Euler’s theorem stipulates that for any positive integer *m* **coprime** with*N,* we have:



Which means that if we raise *any* positive integer *m* coprime with *N* to the power *φ(N)* then divide by *N*, the remainder of the division will be equal to 1. Thus, for **any** *positive* integer *k* we have:

Multiplying each side by *m*we get:



The formula  ***k•φ(N) + 1***reminds **Bezout’s** **identity**, which states that for every number ‘*e’* **coprime** with *φ(N)*, there are two integers *d* and *k* such that:

<https://en.wikipedia.org/wiki/B%C3%A9zout%27s_identity>

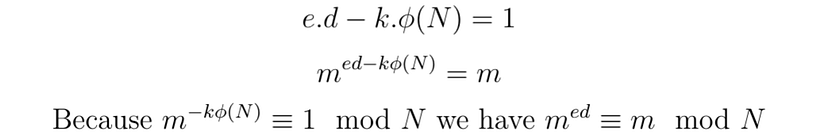
By replacing in the formula above, we get:

Which can also be written as:



By considering ***f*** the function that maps *x* to ***f(x)=x^emodN,*** and ***g*** mapping *x* to ***g(x) = x^d mod N,*** where *mod* returns the remainder of the division, we have ***g(f(m)) = m*** for every *m < N*.

The careful reader will note that the *k* appearing in Bezout’s equality is an integer that may or may not be positive. However, only a positive *k* gives ***m^kφ(N)*≡*1 mod N***. In the case of a negative *k*, we just move ***k.φ(N)*** to the other part of the equation:



We now have two functions ***f***and ***g*** which give us the initial integer *m* when we compute ***g(f(m))***. The integer *m* will act as the message to send. ***f*** will be made public, and ***g*** will be private. Meaning that the integers *e* and *N* will be public, and the integer ***d*** is private. The couple *(e, N)* is the public key, and the couple *(d, N)* is the private key. **We need to choose *N* such that it is infeasible to compute *d* knowing just *e* and *N***.

**Choice of N**

To derive *d* from *e*, an attacker will have to compute *φ(N)* first.

If we choose *N* to be a prime number, *φ(N)* is simply equal to *N-1* because all numbers between *1* and *N-1* are coprime with *N*. In this case, because *N* is public, *φ(N)* is known with no effort to the attacker.

How about *N* the square of a prime number *p*: ***N = p²*** ? Well, It can be easily shown that ***φ(p²) = p(p-1)***. Thus, knowing *p* is enough to know *φ(N)*. *p* can be obtained in a logarithmic time complexity by performing a binary search between *1* and *N*.

This can be easily improved by choosing *N* a product of two *different* prime numbers *p* and *q*:***N = p.q.*** In this case:



One should know *p*and *q* in order to compute *φ(N)*. The time complexity to compute *p* and *q* is more than linear with respect to *N*. If the attacker reaches the colossal speed of testing over 10²⁰ numbers per second (imagine 1000 companies in the size of Google, using all their servers for this search, each server performing at a rate of 10 billion numbers per second), it will still take them over 10¹⁹⁰ centuries to derive *p* and *q* if *p* and *q* are chosen big (for example 2048 bits wide each, which is the size in use today). So choosing ***N=p.q*** the product of two primes is enough to secure *RSA*.