Julian Sequeira

Shayan Ghazi

OpenSSL Tutorial

---

1. Introduction

OpenSSL is an open source software library useful for encryption and secure network communication. SSL stands for **S**ecure **S**ockets **L**ayer, a cryptographic communications protocol. This tutorial will demonstrate one way to use openssl to exchange a file between two parties. There are 3 sections on asymmetric encryption, certificates and RSA respectively- please feel free to skip straight to section 5 if you already have a good understanding of them.

---

2. Asymmetric Encryption

Let's say you have two parties (Alice and Bob) communicating over an insecure network, one on which an attacker (Eve) could intercept all communications. They want to communicate with **confidentiality**, meaning Eve shouldn't be able to understand what they're talking about, even if she can listen to everything they are sending each other. They want **authenticity**- Eve shouldn't be able to defraud Alice by sending messages to her claiming to be from Bob, and vice versa.

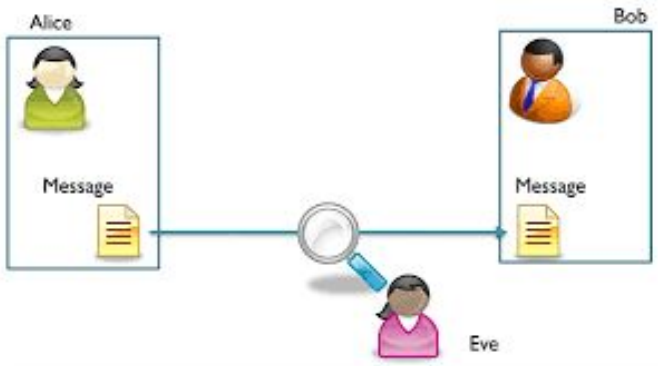

Figure 2.1: Eve listening in on Alice and Bob's connection (src)

You might think to yourself- "Easy! Alice can simply encrypt her message with a key before sending it to Bob. And then Bob can decrypt the message with that key. Eve won't be able to read anything!". But that implies Alice and Bob must both possess the same key, and that begs the question- *how did they securely exchange a symmetric key?*

Alice and Bob can meet up in person to exchange the key- but this isn't always practical in today's interconnected age. Think of Alice as an internet user and Bob as a webserver- imagine how inconvenient it would be if you had to visit a server farm everytime you wanted to access a new website!

One solution here is to use asymmetric encryption. Alice can generate a pair of keys- a **private key** that she keeps to herself, and a **public key** that is freely available to anyone who wants it. If someone encrypts some data with her public key, it can be decrypted with her private key. And she can "sign" data with her private key- a signature that can then be verified with her public key. Bob must have his own set of public and private keys that behave in the same fashion.

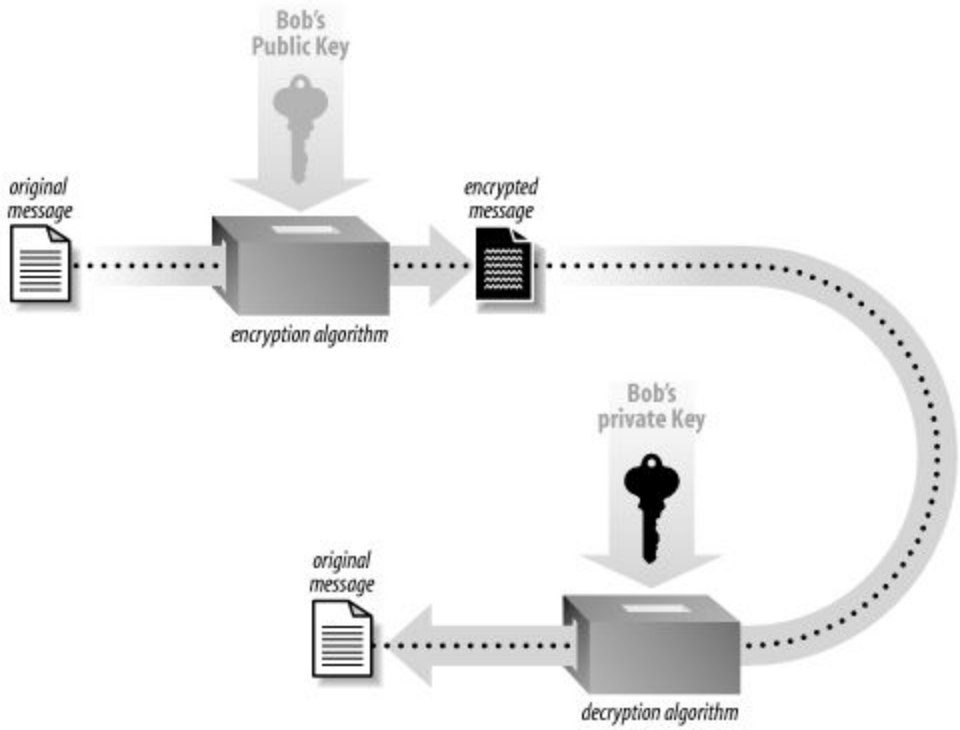

Figure 2.2: Encryption using Bob's public key (src)

So if Alice wants to send Bob a message in a confidential manner- she can encrypt it with Bob's public key. Only Bob's private key can be used to decrypt it, hence only Bob can read the message. And if Alice wants to authenticate herself to Bob, she can generate a signature by signing the message with her private key.

Note: In the RSA cryptosystem, signing a message involves hashing it and then encrypting the hash using a private key. Verifying that signature consists of decrypting the ciphertext using the corresponding public key and checking it against a hash of the message.

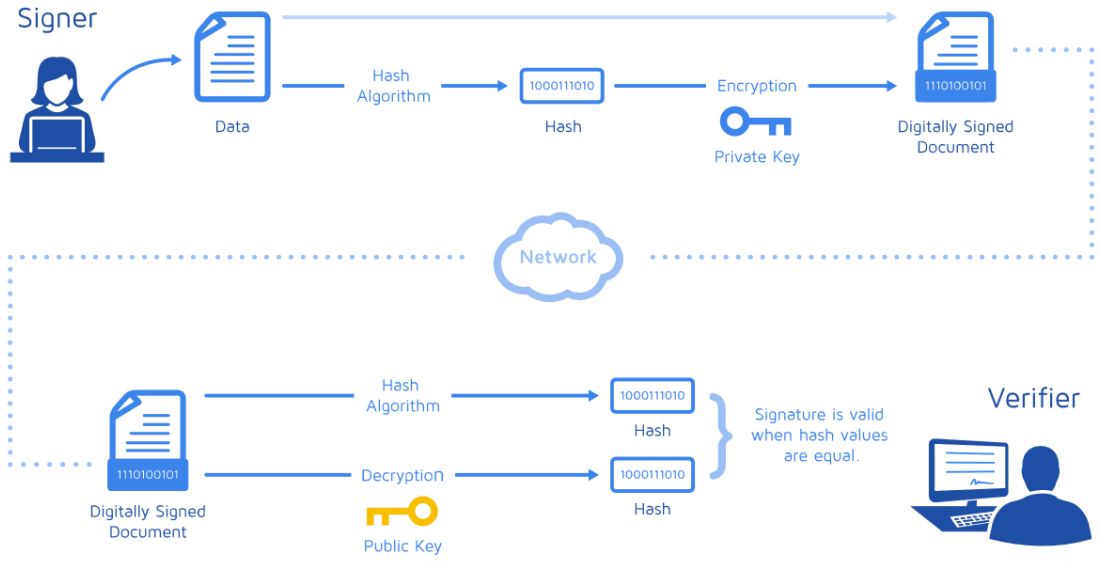

Figure 2.3: Signing a document using a private key (src)

Here's an example of a protocol that ensures both authenticity and confidentiality:

1. Alice generates ciphertext by encrypting her message with Bob's public key
2. Alice generates a signature by hashing her message and then encrypting it with her private key
    - Note: If she didn't hash the message, Eve could simply decrypt this signature using Alice's freely available public key, and thus read the message
3. Alice sends both the ciphertext and signature to Bob
4. Bob decrypts the ciphertext into the message with his private key
5. Bob decrypts the signature using Alice's public key
6. Bob compares the decrypted signature to a hash of the message- if they match then he knows that only Alice could have sent the data

You might be wondering- how do Alice and Bob exchange public keys? Can't they be tampered with or falsified by Eve, just like all their communications? The answer is in the next section...

3. Certificates

Alice and Bob can use **certificates** to ensure that their public keys are authentic. They first generate a **certificate signing request (CSR)** that contains their public keys, some important information about themselves (name, address, organization, etc..) and is signed with their private keys. These CSR's are then signed by a trusted third party- a **Certificate Authority (CA)**, with a private key of its own, and turned into certificates.

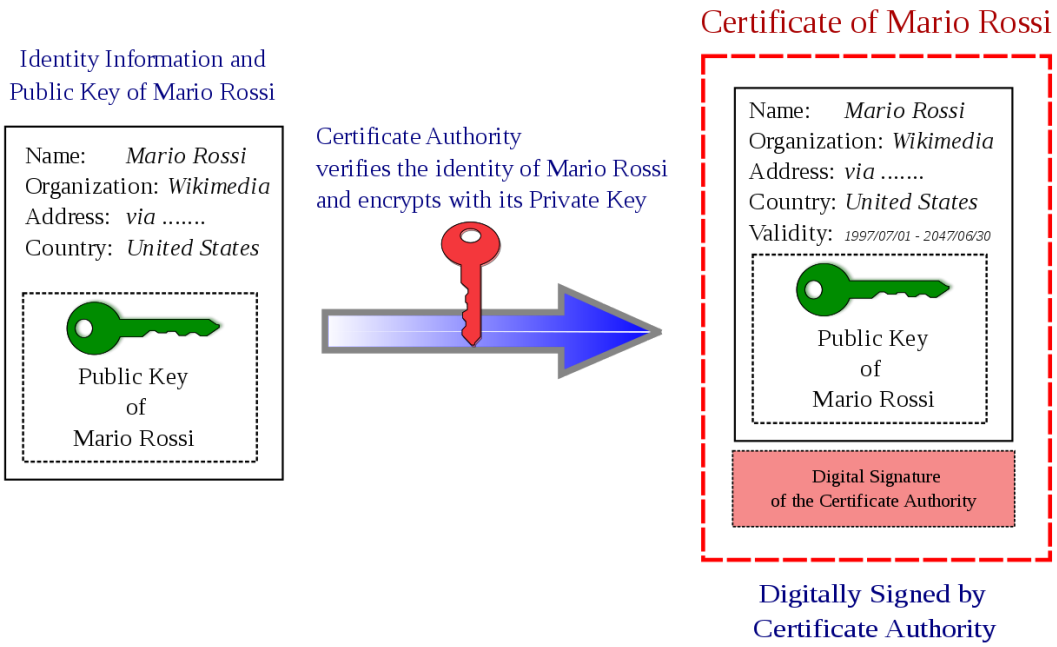

Figure 3.1: A certificate signed by a CA (src)

The CA also has a self-signed certificate, and Alice can verify Bob's certificate by checking it against the CA's certificate. She doesn't have to worry about the authenticity of the CA's certificate because it isn't sent to her over a network. Instead, she'll already have the CA's certificate on her computer (most browsers come with a list of pre-installed certificates from trusted CA organizations, as does Windows).

Each CA is an established organization with a reputation. It does all the legwork in determining that Bob is who he says he is, and in making sure that its private key is not compromised. It charges Bob for this service (although there are some free options like Let's Encrypt). In this way Alice can trust the CA certificates that are on her computer- she doesn't have to worry about the specifics behind how Bob and the CA are communicating.

You can check out wikipedia for a list of the largest certificate authorities by market share.

Let's take a look at the structure of an actual certificate. We're going to see what ssl2buy.com's certificate looks like- you can follow along yourself if you'd like by visiting this page.

Chrome Developer Tools → Security → View Certificate

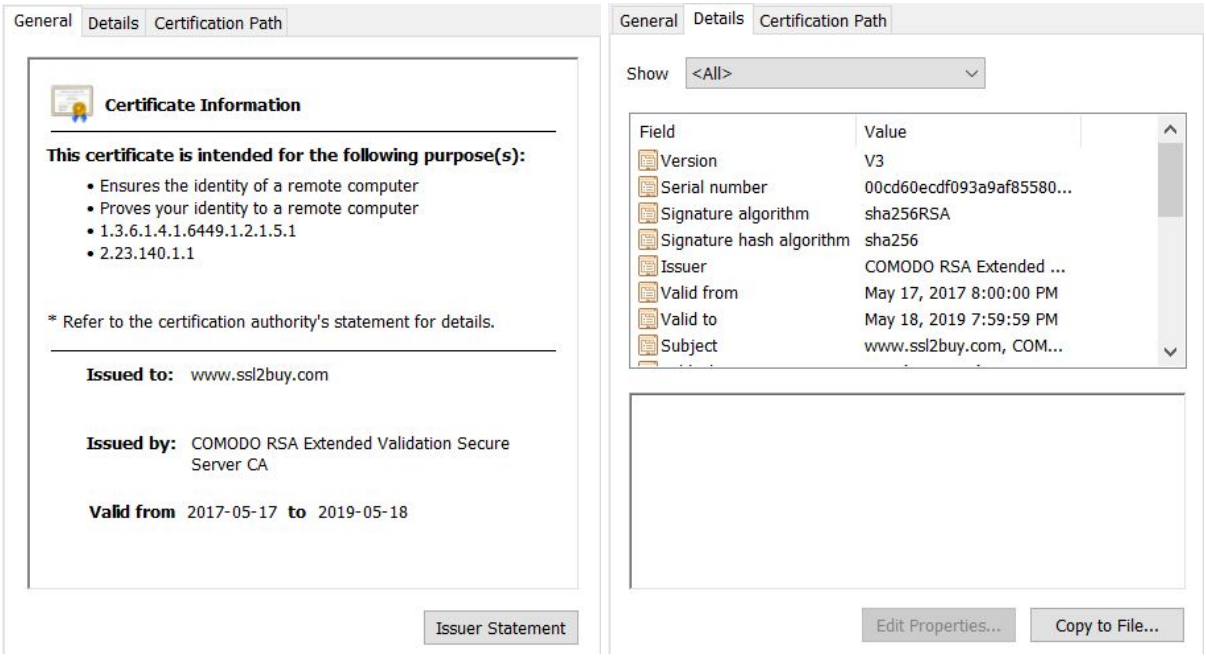

Figure 3.2: A certificate as shown by Chrome

Above we can see that www.ssl2buy.com has been issued a certificate by COMODO RSA Extended Validation Secure Server CA. Note the validity period- the CA also inputs an expiry date after which the certificate is no longer valid. On the right we can see that the signature algorithm is **sha256RSA** - this means that the CA produced a signature by hashing the rest of the certificate with sha256, then encrypting that hash using RSA and its private key.

Chain of trust

The Certificate Authority's job involves signing many different signing requests with its private key. A single key also means a single point of failure- if it's compromised, then all certificates signed by that CA cannot be trusted. And if the CA wants to use one private key for all it's servers, it must copy the key onto every one of them, increasing it's vulnerability.

Instead, CA's use a hierarchical structure of certificates and private keys- a "root" key signs a number of "intermediate" certificates with their own intermediate private keys. The intermediate keys can be used to sign lower-level certificates, and so on, for as many levels as required. The intermediate or low level keys are actually used for signing end-entity CSRs, not the root key. The root key is kept in some secure, encrypted, offline environment, never to be used unless the CA wants to generate more intermediate certificates.

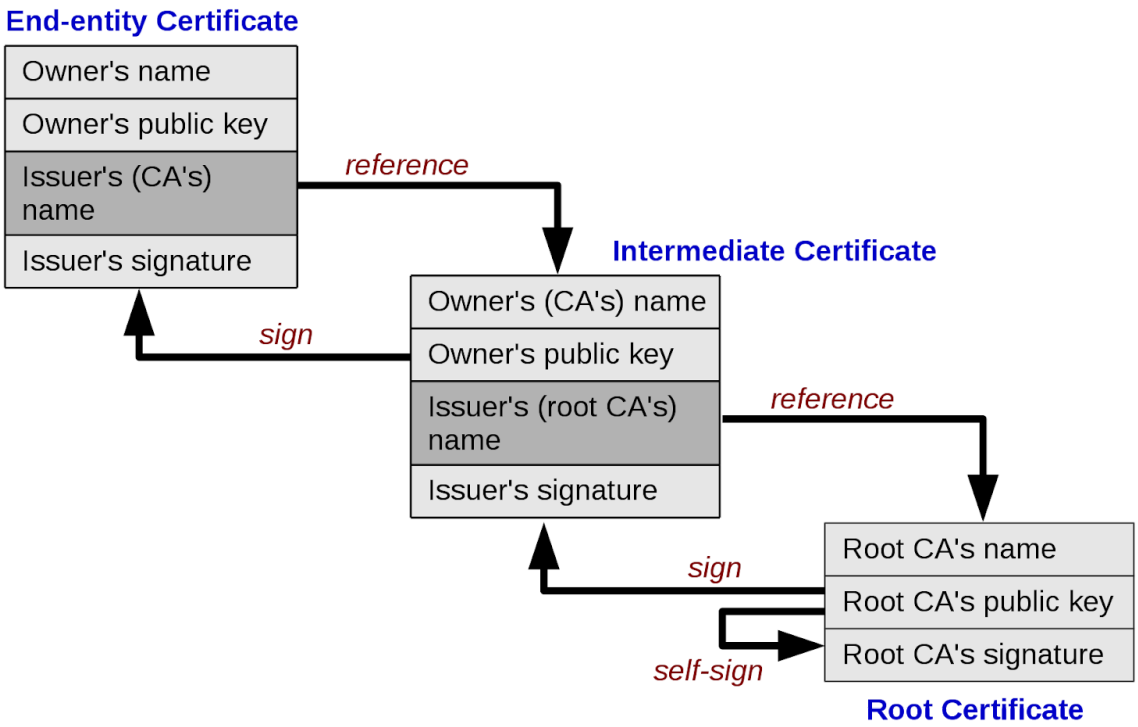

Figure 3.3: An example of a certificate hierarchy (src)

So using the hierarchy above, if Alice wants to verify Bob's certificate, she'll first check it against the Intermediate Certificate. If she already has the intermediate certificate in her computer's store of trusted certificates, she's done. If not, she has to check the intermediate certificate against the root certificate. This is why it's called a **chain of trust** -if at any point any of these verification 'links' are broken, Bob's certificate can not be considered valid.

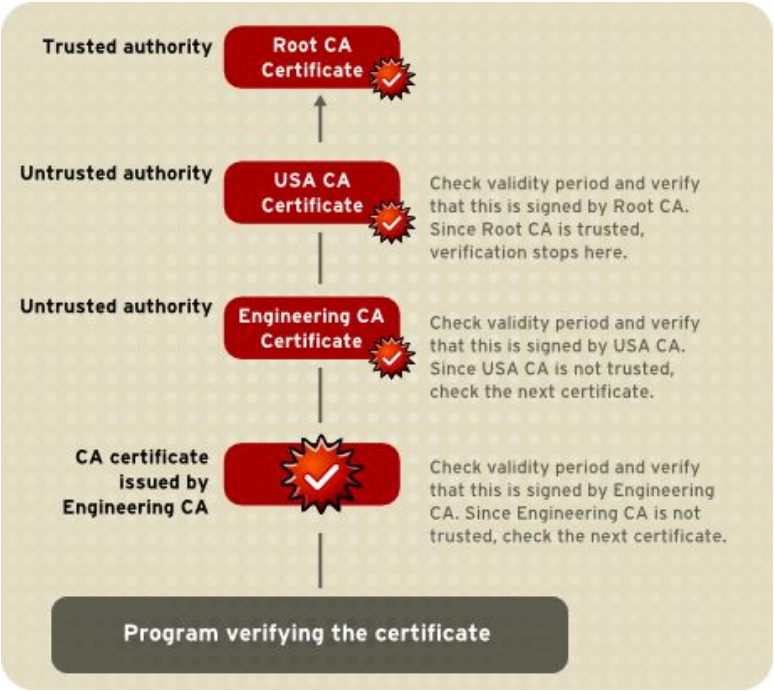

Figure 3.4: Verifying a chain of certificates (src)

We can take a look at ssl2buy.com's certificate and see that it also relies on a chain of trust:

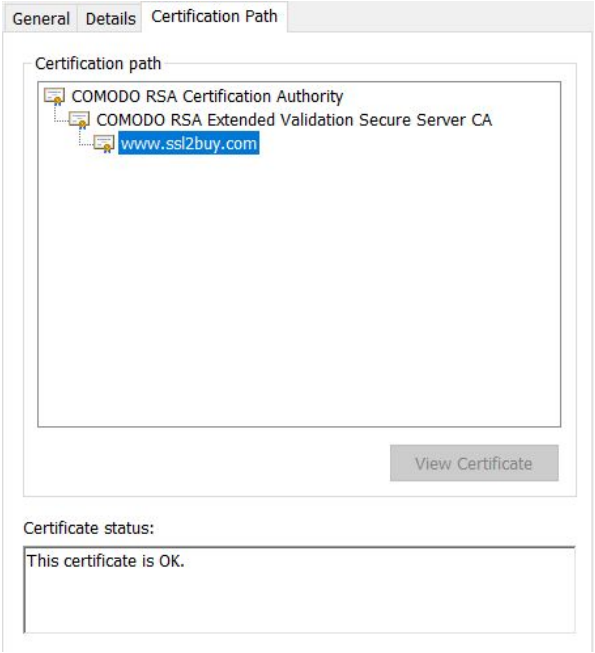

Figure 3.5: A certification path

CA's can also maintain certification revocation lists- certificates that have been revoked before their expiration date and should no longer be trusted. All this is part of a **public-key infrastructure** that Alice and Bob can use to communicate.

Finally, let's take a look at ssl2buy's public key (as contained in their certificate):

```
Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:df:50:d5:9b:44:99:8a:27:5d:af:b0:31:7a:ef:
                    35:da:12:cd:c0:52:2d:2d:02:51:e9:d7:d0:d2:9d:
                    d0:4d:6c:1b:4d:35:fa:a7:23:65:64:2d:92:ed:5e:
                    f1:91:c6:37:5f:71:12:77:f3:a4:91:86:fa:3f:4d:
                    ea:af:4a:b1:22:ff:be:2f:83:26:91:f0:c3:47:83:
                    09:c4:63:7f:d2:ae:e7:79:46:0b:dd:46:04:20:e8:
                    30:00:df:5e:06:26:3c:24:28:6e:a0:c1:b2:d4:26:
                    83:27:76:72:25:4c:1e:09:fb:b3:1f:60:d5:64:02:
                    74:2d:e4:31:76:b8:64:54:11:d9:23:76:1f:4f:10:
                    0d:ee:3f:9f:23:3f:2f:b2:01:74:9e:49:68:29:d9:
                    35:70:1a:2f:74:42:ca:10:58:fb:b1:f5:da:1c:dd:
                    87:ba:05:4c:61:5e:1b:7b:46:65:b2:6b:e2:61:dc:
                    50:a9:73:cc:ae:5a:6e:6b:2e:c3:18:3b:1e:bf:58:
                    d9:f7:4f:d0:16:98:53:b3:5b:51:52:66:23:43:be:
                    e2:5c:7d:67:ff:cc:90:55:51:a7:d3:96:b6:fa:fa:
                    da:e3:83:39:6e:57:ed:81:d8:04:1d:35:b6:dc:a8:
                    7a:f9:2c:f5:39:c5:c5:2f:0c:d7:cc:0c:f2:f3:74:
                    9d:db
                Exponent: 65537 (0x10001)
```

What does *modulus* or *exponent* mean? Find out in the next section…

4. RSA

The setup behind RSA is as follows:

1. Select two large primes p and q
2. Compute $n = pq$ and $\Phi(n) = (p - 1)(q - 1)$
3. Choose an encryption key e that is relatively prime to $\Phi(n)$
4. Calculate a decryption key d such that $ed = 1 \bmod \Phi(n)$
5. Your private key is all of $\{e, n, p, q, d\}$
6. Your public key is $\{e, n\}$ i.e the **public exponent** and **modulus**

Then the encryption and decryption functions are (where M = message, C = ciphertext):

- $C = M^e \bmod n$
- $M = C^d \bmod n$

It's important to note that in practice M should not be a plaintext message. Rather a plaintext message should be processed into M using a padding algorithm. You can read more here.

So why does RSA work?

- **Prime generation is easy** - it's easy to find a random prime number, even large ones
- **Multiplication is easy** - given p and q, it's easy to calculate $n = pq$
- **Modulo inverse is easy** - given e and $\phi(n)$, easy to calculate d so that $ed \bmod \phi(n) = 1$
- **Modular exponentiation is easy** - given n, m and e, it's easy to compute $c = m^e \bmod n$
- **Prime factorization is hard** - given n it's hard to find primes p and q such that $pq = n$
- **Modular root extraction is hard** - given n, e and c, it's difficult to recover m such that $c = m^e \bmod n$, without knowing p or q (or d).

Calculating m^e can take some time, so we want to choose an e that makes this simpler. So often e is picked first, and then the primes p and q are chosen such that $n = pq$ and $gcd(e, \phi(n)) = 1$. In openssl, the default value of e is 65,537. This is a **fermat prime**, it is of the form 2^(2^k) + 1, where $k = 4$. So to calculate m^e, all you have to do is square m, $k+1$ times and then multiply that number by m. (CPUs work in binary and so are very efficient at squaring numbers)

Note: RSA cannot encrypt data larger than the modulus length. It's not meant to encrypt arbitrarily large files. You could of course break the data into small chunks and encrypt those chunks separately, even using techniques to combat frequency analysis. But RSA is still relatively computationally expensive- you're better off using it to exchange a symmetric key and go from there.

This was a very brief intro to RSA. If you're interested in learning more, see this tutorial.

5. The Protocol

So, given what we now know about asymmetric encryption, certificates and RSA, let's put it together in a single protocol:

1. Alice needs a certificate:
    a. She chooses a public exponent *e* and generates a private key
    b. She generates a public key from that private key
    c. She generate a certificate signing request → sends that to the CA
    d. The CA generates a certificate for Alice and signs it → sends it to Alice
2. Alice gets Bob's certificate from him:
    a. She verifies it using the CA's certificate (already pre-installed on her computer)
    b. She extracts Bob's public key from it
    c. She attempts to encrypt her large message using Bob's public key → error!
3. Alice picks a symmetric key:
    a. She picks a strong symmetric key using a pseudo-random number generator
    b. She encrypts it with Bob's public key → symkey.enc
    c. She hashes it and then encrypts it with her private key → signature.bin
    d. She sends both symkey.enc and signature.bin to Bob
4. Bob deciphers and verifies the symmetric key:
    a. He decrypts symkey.enc using his private key
    b. He gets and verifies Alice's certificate and extracts her public key
    c. He decrypts signature.bin using Alice's public key
    d. He compares a hashed symkey with the decrypted signature, they must match
5. Alice encrypts her large message with that symmetric key:
    a. She needs to use choose a symmetric key encryption algorithm
    b. Bob decrypts using the symmetric key and that same algorithm
    c. She can also use something like HMAC now for authentication (this won't be covered in this document but it's similar to how she creates her signature, and you can read more here)

Note: In real life, the protocols used are a little more complicated than this. You'll notice that both parties need to be using the same hashing and encryption algorithms, requiring more initial communication (this is done in the TLS handshake for example).

---

6. OpenSSL Demo

Here we'll implement all the steps of that protocol, using openssl terminal commands. In practice you're more likely to use openssl in the form of an API in another language- but learning the terminal commands is still valuable as a transferable skill. Each command is displayed with some explanations of its flags below.

If you want to follow along, you can make 3 folders, 1 for Alice, Bob and the CA respectively. You need to repeat steps 1.a and 1.b for Bob and CA so they can have their own pair of keys. And you need to generate a self-signed certificate for the CA (shown below).

Step 1.a - Alice generates a private key

```
openssl genpkey -algorithm RSA -pkeyopt rsa_keygen_bits:2048 -pkeyopt
rsa_keygen_pubexp:3 -out privkey-A.pem
```

- **genpkey** → generate a private key
- **-algorithm RSA** → use the RSA algorithm (can also take "EC" for elliptic-curve)
- **-pkeyopt opt:value** → set opt to value (see items below)
- **rsa_keygen_bits:2048** → sets the size of the key to 2048 bits (the default is 1024)
- **rsa_keygen_pubexp:3** → sets the public exponent *e* to 3 (default is 65, 537)
- **-out privkey-A.pem** → outputs to the file privkey-A.pem

Step 1.b - Alice generates a public key

```
openssl pkey -in privkey-A.pem -pubout -out pubkey-A.pem
```

- **pkey** → processes public or private keys
- **-in privkey-A.pem** → read the key from filename privkey-A.pem
- **-pubout** → output a public key (by default, a private key is output)

Aside: viewing the keys in plain text

The keys are saved in base64, and aren't human readable if you open them in a text editor or the terminal. Luckily, openssl provides us with a handy set of commands to convert them to text. The (**-noout**) flag suppresses the command from printing out the base64 encoding as well.

```
openssl pkey -in privkey-A.pem -text -noout
openssl pkey -pubin -in pubkey-A.pem -text -noout
```

Note the size difference in the private key and the public key (one is a subset of the other, afterall). There are some additional values stored in the private key that you won't recognize (exponent1, exponent2 and coefficient). These are stored by openssl to speed up decryption.

Step 1.c - Alice generates a certificate signing request

```
openssl req -new -key privkey-A.pem -out A-req.csr
```

- **req** → creates and processes signing requests
- **-new** → generates a new certificate request, will prompt Alice for some information
- **-key privkey-A.pem** → signs the request with Alice's private key

The command will prompt Alice with these questions:

- **C**ountry code [**C**]: {Alice fills in her country code}
- Province/**ST**ate name [**ST**]: {Alice fills in her province name fully}
- City/**L**ocation [**L**]: {The city Alice's business is registered in, for example}
- **O**rganization Name [**O**]: {Alice's business name, for example}
- **O**rganizational **U**nit Name [**OU**]: (Optional) {What part of the company is she?}
- **C**ommon **N**ame [**CN**]: the hostname+domain, i.e. "www.alice.com"
- A challenge password []: {this can be used as a secret nonce between Alice and CA}

Aside: generating a self-signed certificate for the CA

```
openssl req -x509 -new -nodes -key rootkey.pem -sha256 -days 1024 -out
root.crt
```

Step 1.d - CA generates and signs a certificate for Alice

```
openssl x509 -req -in A-req.csr -CA root.crt -CAkey rootkey.pem
-CAcreateserial -out A.crt -days 500 -sha256
```

- **x509** → an x509 certificate utility (displays, converts, edits and signs x509 certificates)
- **-req** → a certificate request is taken as input (default is a certificate)
- **-CA root.crt** → specifies the CA certificate to be used as the issuer of Alice's certificate
- **-CAkey rootkey.pem** → specifies the private key used in signing (rootkey.pem)
- **-CAcreateserial** → creates a serial number file which contains a counter for how many certificates were signed by this CA
- **-days 500** → sets Alice's certificate to expire in 500 days
- **-sha256** → specifies the hashing algorithm to be used for the certificate's signature

Aside: viewing the certificate as text

```
openssl x509 -in Alice.crt -text -noout
```

Step 2.a - Alice verifies Bob's public certificate

```
openssl verify -CAfile root.crt Bob.crt
```

- **verify** → a utility that verifies certificate chains
- **-CAfile root.crt** → specified the trusted certificate (root.crt)
- **Bob.crt** → the certificate to verify
- If you get an OK, you know the certificate can be trusted

Step 2.b - Alice extracts Bob's public key

```
openssl x509 -pubkey -in Bob.crt -noout > pubkey-B.pem
```

- **-pubkey** → outputs the certificate's public key (in PEM format)

Step 2.c - Alice tries to encrypt her largefile.txt with Bob's public key

```
openssl pkeyutl -encrypt -in largefile.txt -pubin -inkey pubkey-B.pem -out
ciphertext.bin
```

- **pkeyutl** → utility to perform public key operations
- **-encrypt** → encrypt the input data
- **error!** (recall: RSA is not meant for encrypting arbitrary large files- Alice needs to use symmetric key encryption for that)

Step 3.a - Alice generates a symmetric key

```
openssl rand -base64 32 -out symkey.pem
```

- **rand** → generates pseudo-random bytes (seeded by default by $HOME/.rnd)
- **-base64 32** → outputs 32 random bytes and encodes it in base64

Step 3.b - Alice encrypts symkey.pem using Bob's public key

```
openssl pkeyutl -encrypt -in symkey.pem -pubin -inkey pubkey-B.pem -out
symkey.enc
```

Step 3.c - Alice hashes symkey.pem and encrypts it using her private key

```
openssl dgst -sha1 -sign privkey-A.pem -out signature.bin symkey.pem
```

- **dgst -sha1** → hash the input file using the sha1 algorithm
- **-sign privkey-A.pem** → sign the hash with the specified private key
- **symkey.pem** → the input file to be hashed

Step 4.a - Bob decrypts symkey.enc using his private key

```
openssl pkeyutl -decrypt -in symkey.enc -inkey privkey-B.pem -out
symkey.pem
```

- **-decrypt** → decrypt the input file

Step 4.b - Bob gets and verifies Alice's certificate and extracts her public key

(This is simply a retread of what Alice did in step 2)

Step 4.c - Bob verifies the message is from Alice

Steps 4.c and 4.d in the protocol are combined in this step. Bob hashes symkey.pem, decrypts signature.bin, and compares the two results in one command:

```
openssl dgst -sha1 -verify pubkey-A.pem -signature signature.bin symkey.pem
```

- **-verify pubkey-A.pem** → verify the signature using the specified filename
- **-signature signature.bin** → specifies the signature to be verified
- **symkey.pem** → the file to be hashed

Step 5.a - Alice encrypts her largefile.txt with the symmetric key

```
openssl enc -aes-256-cbc -pass file:symkey.pem -p -md sha256 -in
largefile.txt -out ciphertext.bin
```

- **enc -aes-256-cbc** → encrypt a file using the aes-256-cbc symmetric key algorithm
- **-pass file:symkey.pem** → specified the file to get the symmetric key from
- **-p** → prints the key, salt, initialization vector to the screen
- **-md sha256** → uses sha256 as part of the key derivation function (a function that derives one or more secondary secret keys from a primary secret key)

Step 5.b - Bob decrypts ciphertext.bin with the same symmetric key

```
openssl enc -aes-256-cbc -d -pass file:symkey.pem -p -md sha256 -in
ciphertext.bin -out largefile.txt
```

- **-d** → decryption flag

---

7. References

https://www.openssl.org/

https://prefetch.net/articles/realworldssl.html

https://jamielinux.com/docs/openssl-certificate-authority/create-the-root-pair.html

https://www.oreilly.com/library/view/network-security-with/059600270X/

http://heartbleed.com/