provide a more scalable solution. The Public Key Infrastructure (PKI) is such an infrastructure. There are two important components in the PKI infrastructure.

- Certificate Authority (CA): They are responsible for verifying the identity of users and providing them with signed digital certificates. The DMV office in our solution basically serves as a CA, but in the real world, this role is taken by companies who have established themselves as a trusted certificate authority.
- Digital Certificates: It is a document that proves the ownership of the public key mentioned in the certificate. It is also called public key certificate. Digital certificate are signed by CAs who certify the ownership of their contained public keys. Therefore, the security of digital certificates is based on the trust placed on CAs.

11.2 Public Key Certificates

A public key certificate basically certifies the ownership of a public key. It consists primarily of a public key and the identity of the owner, along with the signature of a trusted party. The recipient can verify the signature to ensure the integrity of a certificate. After a successful verification, the recipient will be sure about the ownership of the public key.

11.2.1 X.509 Digital Certificate

The format for public key certificates is defined by the X.509 standard [Cooper et al., 2008]. We use a certificate from Paypal to show the general structure of an X.509 digital certificate.

Listing 11.1: `Paypal`'s X.509 certificate

```
Certificate:
Data:
  Serial Number:
        2c:d1:95:10:54:37:d0:de:4a:39:20:05:6a:f6:c2:7f
  Signature Algorithm: sha256WithRSAEncryption
  Issuer: C=US, O=DigiCert Inc, OU=www.digicert.com,
        CN=DigiCert SHA2 Extended Validation Server CA
  Validity
     Not Before: Aug 14 00:00:00 2018 GMT
     Not After : Aug 18 12:00:00 2020 GMT
  Subject: businessCategory=Private/Organization/
        jurisdictionC=US/
        jurisdictionST=Delaware/
        serialNumber=3014267, C=US, ST=California, L=San Jose,
        O=PayPal, Inc., OU=CDN Support, CN=www.paypal.com
  Subject Public Key Info:
     Public Key Algorithm: rsaEncryption
        Public-Key: (2048 bit)
        Modulus:
          00:ce:a1:fa:e0:19:8b:d7:8d:51:c7:d5:62:84:83:
          13:b9:d7:f6:cd:93:c5:70:d1:69:59:03:2b:b4:8b:
          ... (omitted) ...
          9c:1a:1c:0a:d5:8a:bd:2c:27:ad:c4:fd:aa:b6:4d:
          bf:7b
```

```
        Exponent: 65537 (0x10001)
Signature Algorithm: sha256WithRSAEncryption
a1:eb:9e:7f:c7:17:2e:28:2f:4d:0b:38:95:bb:5b:ca:9e:14:
38:8c:ec:a6:23:26:1f:3b:6a:07:de:4e:4b:41:11:fe:ee:fd:
... (omitted) ...
71:2e:bd:cb
```

- Issuer: this field contains the information about the CA who issues the certificate. In the above example, `DigiCert` is the one who issues the certificate.
- Subject: this field contains the owner's information, i.e., who owns the public key contained in the certificate. In the above example, `Paypal` is the rightful owner of this certificate. The `Subject` field is a very important field, because that is the main purpose of digital certificates, which certify that the enclosed public key belongs to the specified subject. Obviously, before issuing a certificate, the CA needs to verify the information in this field. More details on the subject field will be discussed later.
- Public key: this field contains the actual public key. In the above example, the field contains a 2048-bit RSA public key, which includes a modulus and an exponent.
- Signature: this field contains the digital signature of the issuer. From the above example, the algorithms used for generating the signature is `Sha256` and `RSA`.
- Validity: this field specifies the validity period of the certificate.
- Serial number: Every certificate has a unique serial number, which distinguishes it from the others. The serial number is at the start of the certificate.
- Extensions: newer versions of X.509 certificates contain optional extension fields (not shown in the example above).

11.2.2 Get Certificate from a Real Server

We can obtain and view certificates from real web servers via browsers or command-line tools. Readers can refer to browser manuals for the browser approach. Here we focus on the command-line approach. The following "`openssl s_client`" command sets up an HTTPS client, and connects to Paypal. The `-showcerts` option tells the command to print out all the certificates received from the Paypal server.

```
$ openssl s_client -showcerts -connect www.paypal.com:443 </dev/null
```

The above command does not print out the actual raw content of the certificate; instead, it prints out the encoded data. An actual X.509 certificate contains binary data, and some of which are not printable characters, making it difficult to print or view. Therefore, when stored in files, X.509 certificates are often encoded using printable characters. A common encoding scheme used by X.509 certificates is Base64; a Base64-encoded X.509 certificate is usually saved in a file with the PEM extension (Privacy Enhanced Mail), enclosed between "—BEGIN CERTIFICATE—" and "—END CERTIFICATE—". When we use the above `openssl` command to print out the certificate from Paypal, the command converts the raw content of Paypal's X.509 certificate into the following PEM content:

```
-----BEGIN CERTIFICATE-----
MIIH2DCCBsCgAwIBAgIQAVvaZl/ES3UXtogsHqvU3DANBgkqhkiG9w0BAQsFADB1
MQswCQYDVQQGEwJVUzEVMBMGA1UEChMMRGlnaUNlcnQgSW5jMRkwFwYDVQQLExB3
... (omitted) ...
5dXMg7vQdcCStA+TLiAV4GxSpqlIVpRF0YpqYbzjTiRne9ak/eG0//m4atvKBpXh
9ZXk76n7dH4/nv2u3h8dbt32AMTVozQCJiMaRlMlMElaNvcPwmGHNnEuvcs=
-----END CERTIFICATE-----
```

We can copy and paste the above PEM content into a file (`paypal.pem`), and run another `openssl` command to decode it. In the following command, we can decode it back to the raw binary data, but we cannot view it using a text editor. We have to use a binary file viewer, such as `ghex`, to view it.

```
$ openssl x509 -in paypal.pem -outform der > paypal.der
```

The best way to view an X.509 certificate is to convert the PEM content into text. The following `openssl` command does that, and the result is what we have seen in Listing 11.1.

```
$ openssl x509 -in paypal.pem -text -noout
```

11.3 Certificate Authority (CA)

A certificate authority is a trusted entity that issues signed digital certificates. Before issuing a certificate, the CA verifies the identity of the certificate applicant. The trust of the Public-Key Infrastructure depends on this verification process. Hence, Certificate Authorities perform this step very strictly. The core functionalities of a CA consist of the following:

1. Verify the subject
2. Issue digitally signed X.509 certificates

Subject verification. An essential component of a digital certificate is the subject field, which contains the certificate owner's identity information. Before signing a certificate, the CA needs to make sure that the person (applicant) applying for the certificate either owns or represents the identity in the subject field. In many public-key certificates, the identity field contains a domain name. CA needs to check whether the applicant owns the domain or not. For example, if an applicant wants to get a certificate for `www.example.com`, to verify whether the applicant owns the domain or not, the CA may give the applicant a randomly generated number, and ask the applicant to put it the website `http://www.example.com/proof.txt`. If the applicant is able to do that, the domain ownership will be verified. Some digital certificates also contain additional information, such as company name and address. Certain special type of CAs also verify this information by checking whether the company has a proper business and legal registration.

Some of the verification can be easily achieved using the publicly available databases or the URL approach described above. For example, domain verification can also be achieved using the public WHOIS database. However, not all information is publicly available, such as whether a company is in a proper legal standing or not. To verify such information, sometimes legal experts or authorities, such as lawyers and government officials, may be involved. Therefore, CAs often delegate the verification functionality to a dedicated entity called Registration Authority (RA).

Signing Digital Certificates. Once a CA has verified the identity information of a certificate request, it can sign the certificate, and thus binds the identity to the public key in the certificate. Signing a digital certificate means that the CA generates a digital signature for the certificate using its private key. Once the signature is applied, the certificate cannot be modified, or the signature will become invalid. The signatures can be verified by anyone who has a copy of the CA's public key.

11.3.1 Being a CA

Let us walk through the entire signing process using a concrete scenario. A bank wants to use the public key technology for its online banking web site `bank32.com`. It needs an X.509 certificate. In the real world, the bank would go to one of the CAs to get such a certificate, and the bank has to pay for it. We are going to emulate that by becoming a certificate authority ourselves. We will use `ModelCA` to refer to this CA in our emulation.

The basic process to apply for a digital certificate is the following: before the process starts, both parties, the bank and `ModelCA`, need to generate their own public-private keys. The bank will then generate a Certificate Signing Request (CSR) containing the domain information and the public key for which it needs to get a certificate. The bank will submit the request to `ModelCA`, who would verify whether the domain `bank32.com` belongs to the bank. For some special certificate types, `ModelCA` would also do an extended verification, such as verifying the bank's business records (more details on this will be given in §11.7.3). Once everything is verified, `ModelCA` will sign the request using its own private key and create a digital certificate. This digital certificate would be given back to the bank, who can use the certificate to set up its HTTPS-based web server. We will follow the entire process step by step using our `Ubuntu16.04` VM.

Step 1: CA's setup. We need a CA to sign certificates. To understand the entire process carried out by CAs, we create our own CA using `openssl`. A default configuration file (`/usr/lib/ssl/openssl.cnf`) is used by `openssl` when signing certificates; the file requires certain folders and files to be set up properly. We first create a directory named as `demoCA`, and then create the following three folders under it: `certs`, `crl`, and `newcerts`. We also need to create the following two files inside `demoCA`: `index.txt` and `serial`. The file `serial` contains the serial number for the next certificate; we can put 1000 or any number in the file as the initial serial number. The commands to achieve the above is given below:

```
$ mkdir demoCA
$ cd demoCA
$ mkdir certs crl newcerts
$ touch index.txt serial
$ echo 1000 > serial
$ cd ..
```

Step 2: Create public/private keys and certificate for `ModelCA`. To verify the certificates created by a CA, we need the CA's public key, so each CA needs a digital certificate of its own. If the CA is an intermediate CA, it needs to get its certificate from another CA. If the CA is a root CA, it generates its own certificate by signing the certificate using its private key. Such a certificate is called *self-signed* certificate; basically the root CA "vouches" for itself. Obviously, such a vouching cannot be trusted. For a root CA's certificate to be trusted, it has to be delivered

to users in a secure manner. For example, when an operating system is installed, it already comes with the certificates from a list of trusted root CAs.

We need to create a public-private key pair for `ModelCA`, and then generate a self-signed certificate for it (in our example, `ModelCA` is a root CA). We can use the following command to achieve that.

```
$ openssl req -x509 -newkey rsa:4096 -sha256 -days 3650
              -keyout modelCA_key.pem -out modelCA_cert.pem
```

The above command generates a public-private key pair (4096-bit RSA keys). The key information, including both public and private keys, is stored in the password-protected file `modelCA_key.pem`; the self-signed certificate is saved in `modelCA_cert.pem`, and it will be valid for 3650 days. It should be noted that in our VM, `openssl` by default uses the SHA1 algorithm to generate one-way hashes, which are then signed with the CA's private key. It was discovered in February 2017 that SHA1 is not secure [Stevens et al., 2017b], so we use the `-sha256` option to switch to the SHA2 algorithm.

Our `ModelCA` is now ready to issue certificates. We show how a bank can get a certificate from this certificate authority.

11.3.2 Getting X.509 Certificate from CA

Assume that a bank wants to set up an HTTPS web server to ensure that customers' interaction with the server is protected. The bank needs to generate a private/public key pair first, apply for an X.509 certificate from a trusted CA, and then deploy the certificate at its web server. To get a hands-on experience with this process, we go through this entire process using `openssl`, `Apache`, and our `ModelCA`.

Step 1: Generate the public/private key pair. To get an X.509 certificate, the bank first needs to generate its own public and private key pair. This can be done using the following `openssl` command.

```
$ openssl genrsa -aes128 -out bank_key.pem 2048
$ more bank_key.pem
-----BEGIN RSA PRIVATE KEY-----
Proc-Type: 4,ENCRYPTED
DEK-Info: AES-128-CBC,E3FA5EAFDD561A61D58F85C0FD21C4E3

YFSFqfP4WNOVTrENW9y5BZYBlpUpoYufE00dzZOpxUdDuL4U7qadAggd+TBTIwBe
...(omitted)...
cFmKGHnwZhGodoo/2PzQ31jmt+IdTeYjuvM+Fa5u4GBYE66YdDYD+OM+WQNFEufw
-----END RSA PRIVATE KEY-----
```

The above command generates a file named `bank_key.pem`, which is protected with a password provided by users during the key generation. The output file `bank_key.pem` is an encoded file. We can use the following command to see its actual content. From the result, we can see that `bank_key.pem` contains both public and private keys. Moreover, it contains the prime factors of the modulus (`prime1` and `prime2`), as well as several other numbers (`exponent1`, `exponent2` and `coefficient`) that are useful for optimizing the decryption process [Menezes et al., 1996].

```
seed@ubuntu:$ openssl rsa -noout -text -in bank_key.pem
Enter pass phrase for bank.key:
Private-Key: (2048 bit)
modulus:
    00:ed:9f:3a:c6:9d:88:d4:fc:23:8a:d2:82:71:d9:
    ......
publicExponent: 65537 (0x10001)
privateExponent:
    57:53:9e:51:21:d2:08:9c:05:1f:de:8f:4b:f1:ff:
    ......
prime1:
    00:fb:49:71:55:39:7f:fd:c8:40:b6:d8:9c:51:0f:
    ......
prime2:
    00:f2:14:2c:b9:ff:ac:d3:44:24:d5:3a:d8:e3:02:
    ......
exponent1:
    40:d6:6c:65:bf:16:65:57:1c:4b:91:8c:93:e5:e4:
    ......
exponent2:
    00:ba:e2:ee:60:ad:cd:1b:d0:d8:ea:b1:22:ad:a6:
    ......
coefficient:
    0a:16:f0:99:b9:19:5f:47:54:ca:6a:c9:04:91:dc:
    ......
```

Step 2: Generate certificate signing request. To get a digital certificate from a CA, the bank needs to create a certificate signing request, which contains the bank's public key and details about its identity, such as organization name, address, and domain name. We can use the following command to generate a certificate signing request based on `bank_key.pem`..

```
$ openssl req -new -key bank_key.pem -out bank.csr -sha256
```

The `openssl` program will ask us to provide the subject information, including company name, address, email, etc. In the common name field (CN), let us use `bank32.com`. The generated certificate signing request is stored in a CSR file `bank.csr`, which is encoded. We can run the following command to see what is actually in a CSR file.

```
$ openssl req -in bank.csr -text -noout
Certificate Request:
    Data:
        Version: 0 (0x0)
        Subject: C=US, ST=New York, L=Syracuse, O=Bank32 Inc.,
            CN=bank32.com/emailAddress=admin@bank32.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                Public-Key: (2048 bit)
                Modulus:
                    00:c1:d9:3f:99:b3:61:fa:00:11:5b:4d:dd:b8:f3:
                    d3:b7:06:d0:84:b2:6f:7e:9c:9b:9a:97:d0:28:e9:
                    ......
```

```
            Exponent: 65537 (0x10001)
    Attributes:
        challengePassword        :unable to print attribute
Signature Algorithm: sha256WithRSAEncryption
     5f:ab:4c:b1:66:b5:03:35:05:e2:cb:4f:c2:7c:ff:88:c8:65:
     27:52:32:77:0b:4c:23:82:8b:25:69:b5:73:1a:16:7c:8e:62:
     ......
```

It should be noted that the signature in the request is generated by the requester using its own private key, i.e., the requester signs its own public key. By verifying the signature using the public key in the request, the CA can be sure that the public key does belong to the requester. The purpose of this signature is to prevent an entity from requesting a bogus certificate of someone else's public key [Nystrom and Kaliski, 2000].

Step 3: Ask CA to sign. In the real world, the bank would submit its CSR file to a CA, who will issue a signed certificate after verifying the information in the CSR. In our emulation, we would send the CSR file to `ModelCA`, who generates a certificate using the following command.

```
$ openssl ca -in bank.csr -out bank_cert.pem -md sha256
             -cert modelCA_cert.pem -keyfile modelCA_key.pem
```

The above command constructs an X.509 certificate using the bank's CSR file (`bank.csr`) and the information from the CA's certificate (`modelCA_cert.pem`); it then uses CA's private key (`modelCA_key.pem`) to sign the certificate. The generated certificate is stored in `bank_cert.pem`. It should be noted again that by default, `openssl` uses SHA1, which is not secure, we need to use the `"-md sha256"` option to force it to use SHA256.

If `OpenSSL` refuses to generate certificates, it is very likely that some of the fields in the request do not match with those of the CA. See the following error message:

```
Using configuration from /usr/lib/ssl/openssl.cnf
Enter pass phrase for modelCA_key.pem:
Check that the request matches the signature
Signature ok
The organizationName field needed to be the same in the
CA certificate (Model CA) and the request (Bank32 Inc.)
```

The matching rules are specified in the default configuration file `openssl.cnf` inside `/usr/lib/ssl/`. By default, we see `"policy = policy_match"`, which requires some of the subject fields in the request to match those of the CA. We can change it to `"policy = policy_anything"`, which is another policy defined in the configuration file and it does not have any restriction. We can also copy the configuration file to our current directory, change this local copy. We run `openssl ca` command again, but this time we use the `-config` option to specify the configuration file. See the following command:

```
$ openssl ca -in bank.csr -out bank_cert.pem -md sha256
             -cert modelCA_cert.pem -keyfile modelCA_key.pem
             -config openssl.cnf
```

11.3.3 Deploying Public Key Certificate in Web Server

After receiving its digital certificate, the bank can deploy the certificate in its HTTPS website. We will first use OpenSSL's built-in server to set up an HTTPS web server, and later we show how to do it for an Apache web server. First, we need to combine the bank's private key and certificate into one file (`bank_all.pem`), and then run the "`openssl s_server`" command to start the server using the public/private keys from `bank_all.pem`. Our server listens to port `4433`.

```
$ cp bank_key.pem bank_all.pem
$ cat bank_cert.pem >> bank_all.pem
$ openssl s_server -cert bank_all.pem -accept 4433 -www
```

The above `openssl` command launches an `openssl` server to handle HTTPS connections. The URL for this web site is `https://bank32.com:4433`. We do need to add the following entry to `/etc/hosts`, mapping the hostname `bank32.com` to IP address `127.0.0.1`, which is `localhost` (for simplicity, we run the server on `localhost`).

```
127.0.0.1   bank32.com
```

Using Firefox browser. Let us fire up the browser and visit `https://bank32.com:4433`, we will see the following error message, indicating that the connection is not secure.

```
Your connection is not secure
...
bank32.com:4433 uses an invalid security certificate.

The certificate is not trusted because the issuer certificate is
unknown. The server might not be sending the appropriate intermediate
certificates. An additional root certificate may need to be imported.

Error code: SEC_ERROR_UNKNOWN_ISSUER
```

This is because the browser does not have `ModelCA`'s public key, so it cannot verify the signature on the bank's certificate. Browsers have a list of trusted CAs, and obviously `ModelCA` is not on that list. If a CA is not trusted, none of the certificates issued by the CA will be trusted. To get on that list, we have to convince whoever developed the browser that `ModelCA` is an trustworthy CA. That is how it works in the real world, but it is impractical for our emulation. Fortunately, the Firefox browser (as well as many other browsers) allows us to manually add a CA's certificate to its trusted list. To achieve that, we can click the following menu sequence:

```
Edit -> Preference -> Privacy & Security -> View Certificates
```

We will see a list of certificates that are already accepted by Firefox. At the `Authorities` tab, we can import our `ModelCA`'s certificate `modelCA_cert.pem`. We need to select the following option: "Trust this CA to identify web sites". We will see that our CA's certificate is now in Firefox's list of the accepted certificates. Now when we visit `https://bank32.com:4433` again, the browser does not show the error message anymore, and we get a reply from the server.

Using "openssl s_client". Instead of using browsers, we can also access a web server using `openssl`'s `s_client` command. This command prints out a lot of debugging information, so it is quite useful to see what actually happens during the interaction between the client and the server. Let us use this command to connect to our web server.

```
$ openssl s_client -connect bank32.com:4433
CONNECTED(00000003)
depth=0 C = US, ST = New York, ..., CN = bank32.com, ...
verify error:num=20:unable to get local issuer certificate
verify return:1
depth=0 C = US, ST = New York, ..., CN = bank32.com, ...
verify error:num=21:unable to verify the first certificate
verify return:1
```

We can see the error messages from the result above. Since the client program do not have the issuer's (`ModelCA`) certificate, it cannot verify the certificate from `bank32.com`. Let us tell the client program about the `ModelCA`'s certificate using the `-CAfile` option. From the result, we can see that there is no more error message.

```
$ openssl s_client -connect bank32.com:4433 -CAfile modelCA_cert.pem
CONNECTED(00000003)
depth=1 C = US, ..., O = Model CA, CN = modelCA.com, ...
verify return:1
depth=0 C = US, ..., O = Bank32 Inc., CN = bank32.com, ...
verify return:1
```

11.3.4 Apache Setup for HTTPS

The HTTPS server setup using `openssl`'s `s_server` command is primarily for debugging and demonstration purposes. Here we show how to set up a real HTTPS web server based on Apache. Assume that the Apache server is already installed (that is the case for our `Ubuntu16.04` VM). We add the following `VirtualHost` entry to Apache's configuration file (`default-ssl.conf`) located in the `/etc/apache2/sites-available/` folder (443 is the default port number for HTTPS).

```
<VirtualHost *:443>
    ServerName bank32.com
    DocumentRoot /var/www/html
    DirectoryIndex index.html

    SSLEngine On
    SSLCertificateFile      /home/seed/cert/bank_cert.pem  ①
    SSLCertificateKeyFile   /home/seed/cert/bank_key.pem   ②
</VirtualHost>
```

In the configuration above, the `ServerName` entry specifies the URL of the web server; the `DocumentRoot` entry specifies where the files of the website are stored. We also need to tell Apache where the server's certificate (Line ①) and private key (Line ②) are stored. After modifying `default-ssl.conf`, we need to run the following commands to start SSL.

```
// Test the Apache configuration file for errors.
$ sudo apachectl configtest
// Enable SSL
$ sudo a2enmod ssl
// Enable the sites specified in default-ssl
$ sudo a2ensite default-ssl
// Restart Apache
$ sudo service apache2 restart
```

Apache will ask us to type the password to decrypt the private key. Once everything is set up properly, we can browse the web site `https://bank32.com:4433`, and all the traffic between the browser and the server will be encrypted.

11.4 Root and Intermediate Certificate Authorities

There are many certificate authorities in the real world, and they are organized in a hierarchical structure as seen in Figure 11.2. CAs at the top of the hierarchy are called root CAs. They can issue certificates directly for customers, or delegate some of the task to their subordinates, which are called intermediate CAs. Intermediate CAs may also further delegate the tasks to their own subordinates.

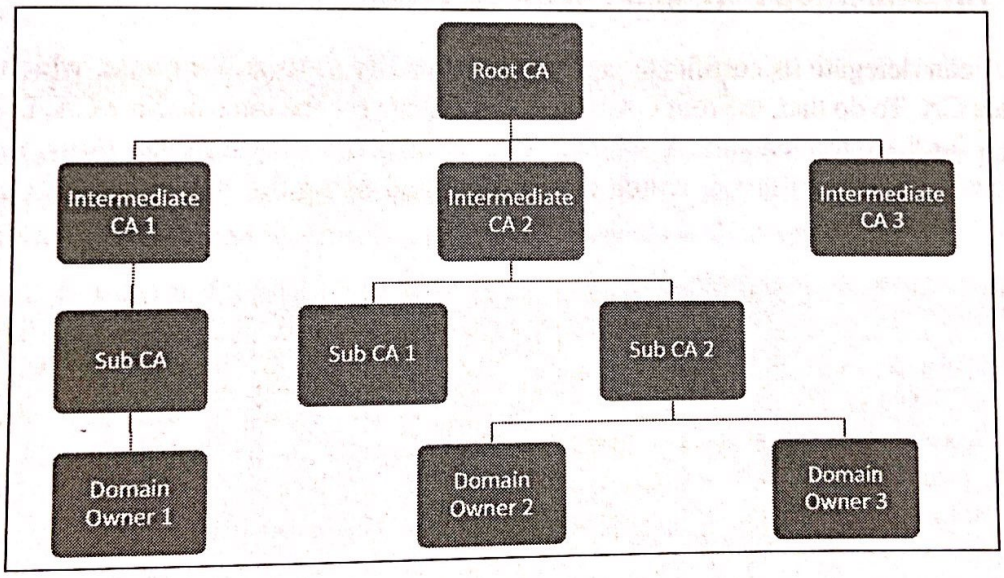

Figure 11.2: Hierarchy Of Certificate Authorities

11.4.1 Root CAs and Self-Signed Certificate

To verify the certificates issued by a root CA, we need to have the root CA's public key. The question is how to get it securely; that is exactly the same question that we are trying to solve with the public-key infrastructure. We cannot ask the root CA to send us its public key because of the man-in-the-middle attack, nor can we find another CA to issue a certificate for the root CA and vouch for it (because it will not be a root CA anymore). Therefore, we have no way to verify whether a public key actually belongs to a root CA or not.

Root CAs' public keys are delivered to users via a different channel. They are pre-installed in the operating systems, browsers, and other software, i.e., the software comes with the public keys of a list of trusted root CAs. It guarantees that these public keys are authentic. Basically, by including these public keys, the software vouches for the authenticity of them. As long as we trust the software, we are trusting the public keys that come with it.

A root CA's public key is also stored in an X.509 certificate, but the certificate is not signed by another CA; instead, it is signed by the CA itself, i.e., it is self-signed. Obviously, the signature on a self-signed certificate does not serve the same purpose as that on a CA-signed certificate, so the trust of the root CAs' public key does not come from the signature. Inside a self-signed X.509 certificate, the entries for the issuer and subject are identical. The following example shows the identical issuer and subject fields in the certificate belonging to VeriSign's root CA.

```
Issuer: C=US, O=VeriSign, Inc., OU=VeriSign Trust Network,
    OU=(c) 2006 VeriSign, Inc. - For authorized use only,
    CN=VeriSign Class 3 Public Primary Certification Authority - G5
Subject: C=US, O=VeriSign, Inc., OU=VeriSign Trust Network,
    OU=(c) 2006 VeriSign, Inc. - For authorized use only,
    CN=VeriSign Class 3 Public Primary Certification Authority - G5
```

11.4.2 Intermediate CAs and Chain of Trust

A Root CA can delegate its certificate issuing functionality to its subordinate, which is called intermediate CA. To do that, the root CA issues a certificate for the intermediate CA, i.e., the root CA vouches for the intermediate CA, who can then issue certificates for other users. Let us look at `Alibaba`'s actual certificate, which can be obtained using the `"openssl s_client"` command.

```
$ openssl s_client -showcerts -connect www.alibaba.com:443

Certificate chain
 0 s:.../O=Alibaba (China) Technology Co., Ltd./CN=*.alibaba.com
   i:.../CN=GlobalSign Organization Validation CA - SHA256 - G2
-----BEGIN CERTIFICATE-----
MIIL+jCCCuKgAwIBAgIMfGvQBaGmvaTO6bIyMA0GCSqGSIb3DQEBCwUAMGYxCzAJ
...
-----END CERTIFICATE-----
 1 s:.../CN=GlobalSign Organization Validation CA - SHA256 - G2
   i:.../O=GlobalSign nv-sa/OU=Root CA/CN=GlobalSign Root CA
-----BEGIN CERTIFICATE-----
MIIEaTCCA1GgAwIBAgILBAAAAAABRE7wQkcwDQYJKoZIhvcNAQELBQAwVzELMAkG
...
-----END CERTIFICATE-----
```

The above result shows a certificate chain obtained from `Alibaba.com`. It contains two certificates. The first one is Alibaba's certificate, issued by a CA called "GlobalSign Organization Validation CA - SHA256 - G2", which is an intermediate CA, and its certificate is the second one in the above result. This certificate is signed by another CA called "GlobalSign Root CA". This is a root CA. To verify the server's certificate, the client (e.g., browser) follows the chain of trust described below.

1. Check whether the root CA who issues the certificate for the intermediate CA is on the browser's trusted CA list. If so, the browser already has the root CA's public key.
2. Verify the intermediate CA's certificate using the root CA's public key.
3. Verify the server's certificate using the intermediate CA's public key.

To see how the verification works, We can manually verify a certificate chain using `openssl`. Let us save the Alibaba's certificate (copy and paste) to a file called `Alibaba.pem`, and save the intermediate CA's certificate to `GlobalSign-G2.pem`. We will also get `GlobalSign Root CA`'s self-signed certificate from a browser, and save it to a file called `GlobalSign-Root.pem`. We then run the following command to verify Alibaba's certificate.

```
$ openssl verify -verbose -CAfile GlobalSignRootCA.pem
                          -untrusted GlobalSign-G2.pem Alibaba.pem
Alibaba.pem: OK
```

In the above command, the `untrusted` option provides a certificate chain, the last of which has to be the domain server's certificate. The `CAfile` option specifies a trusted CA certificate (must be self-signed), which is used to verify the first certificate in the chain. After the first certificate is verified, it is used to verify the second certificate, and so on. If the entire chain is verified successfully, `OK` will be printed out.

11.4.3 Creating Certificates for Intermediate CA

A CA can issue a certificate for an intermediate CA using `openssl`. The intermediate CA needs to generate a certificate request (`modelIntCA.csr`), sends it to a trusted CA, who can use the following command to generate a certificate based on the request.

```
// Intermediate CA generates public/private key pair
$ openssl genrsa -aes128 -out modelIntCA_key.pem 2048

// Internediate CA generates certificate request
$ openssl req -new -key modelIntCA_key.pem -out modelIntCA.csr
              -sha256

// Root CA (ModelCA) issues a certificate to ModelIntCA
$ openssl ca -in modelIntCA.csr -out modelIntCA_cert.pem -md sha256
             -cert modelCA_cert.pem -keyfile modelCA_key.pem
             -extensions v3_ca
```

The above certificate-issuing command is quite similar to the one used in issuing a certificate for a server, except that it includes the `"-extensions v3_ca"` option. This option tells `openssl` to set the CA entry in the certificate's extension field to TRUE, indicating that the certificate belongs to an intermediate CA, and it can be used to verify other certificates issued by this intermediate CA. If we look at the extension field of the certificate, we will see the following.

```
X509v3 extensions:
  X509v3 Basic Constraints:
      CA:TRUE
```

The CA field in a non-CA certificate has the value FALSE, which means that the certificate cannot be used to verify other certificates (i.e., the owner of the certificate cannot serve as a CA). Even if the owner of this non-CA certificate issues a certificate for some one, this issued certificate cannot be verified because the issuer's certificate is not a CA certificate.

11.4.4 Apache Setup

If a web server's certificate is signed using an intermediate CA, when a client (browser) asks for its certificate, it should send out the certificates of all the involved intermediate CAs, in addition to its own certificate. These certificates should be saved in a single file, and the file name should be added to the Apache configuration file `default-ssl.conf` (in the `SSLCertificateFile` field). See the following configuration:

```
<VirtualHost *:443>
    ServerName bank32.com
    DocumentRoot /var/www/html
    DirectoryIndex index.html

    SSLEngine On
    SSLCertificateFile      /home/seed/cert/bank_cert2.pem
    SSLCertificateKeyFile   /home/seed/cert/bank_key.pem
</VirtualHost>
```

In the configuration above, `bank_cert2.pem` contains two certificates: the first is `bank32`'s, and the second is `modelIntCA`'s. The order is important.

11.4.5 Trusted CAs in the Real World

There are many trusted CAs, including root CAs and intermediate CAs. According to a report published by W3Techs in April 2017 [W3Techs, 2017], Comodo takes most of the market share (42.2%), followed by IdenTrust (25.2%), Symantec Group (15.0%), GoDaddy Group (7.6%), GlobalSign (4.8%), and DigiCert (2.3%). Not all of the trusted CAs are present in all browsers. We can get the list of the trusted CAs supported by browsers using the following instructions.

For the Chrome browser (version 70.0.3538.77):

1. Go to the `Settings` page by either finding the menu item from Chrome's menu, or type `chrome://settings` in the URL field. Once we are in the settings page, scroll to the bottom, and click the `Advanced` link.
2. Find and click the `"Manage Certificates"` button. A pop-up window will appear.
3. On this window, we can see several tabs. There is one tab for root CAs and another for intermediate CAs. Click on these tabs, and we can see a list of trusted root and intermediate CAs, respectively.

For the Firefox browser (version 60.0b10):

1. Type `about:preferences` in the URL field, and we will enter the setting page.