Public Key Cryptography: key formats

In this short write-up I'm writing down what I know about the format of keys for public key cryptography. There are, unfortunately, a number of formats and possibilities and we will only scratch the surface here.

The main complication is with public keys. There are at least 5 formats that I've seen. Which one you get depends on how the key is produced.

Format A: `ssh-rsa`

This format is what `ssh-keygen` gives. It is sometimes called Open SSH (`en.wikipedia.org/wiki/Ssh-keygen`), and is distinguished by the text `ssh-rsa` in ASCII encoding at the beginning of the public key file.

To generate such a key from the command line (with `openssh` installed):

```
ssh-keygen -t rsa -C "te" -f ./kf
```

According to the man page:

> The type of key to be generated is specified with the -t option. If invoked without any arguments, ssh-keygen will generate an RSA key for use in SSH protocol 2 connections.

The default size is 2048, and the $-C$ flag indicates a comment is to be added to the key.

One can generate a key that is encrypted, with a passphrase to protect it or not. Just follow the prompts. The private key looks like this:

```
$ hexdump -C kf 00000000 2d 2d 2d 2d 2d 42 45 47 49 4e 20 52 53 41 20 50
|-----BEGIN RSA P| 00000010 52 49 56 41 54 45 20 4b 45 59 2d 2d 2d 2d 2d
0a |RIVATE KEY-----.| 00000020 4d 49 49 45 6f 67 49 42 41 41 4b 43 41 51
45 41 |MIIEogIBAAKCAQEA| ..
```

We'll deal with this format in a minute.

The public key looks like this:

```
$ hexdump -C kf.pub
00000000 73 73 68 2d 72 73 61 20 41 41 41 41 42 33 4e 7a |ssh-rsa AAAAB3Nz|
00000010 61 43 31 79 63 32 45 41 41 41 41 44 41 51 41 42 |aC1yc2EAAAADAQAB|
00000020 41 41 41 42 41 51 43 2b 62 5a 6c 47 71 6e 72 48 |AAABAQC+bZlGqnrH|
..
```

The part up to and including `AAABAQ` in the third line is common to all such keys. The data after the first 8 characters is encoded in base64.

The file ends with the comment `" te"` (without quotes) in ASCII encoding, separated by a space from the key data.

Format B: `openssl`

This format is distinguished by the text `BEGIN PUBLIC KEY`, (i.e. missing the word `RSA`). It is called X.509, according to

`www.cryptosys.net/pki/rsakeyformats.html`

and is the same as PKCS #1 (PKCS #1/X.509), according to

`www.cryptopp.com/wiki/Keys_and_Formats`

It is also called Open SSL format and it is what `openssl` generates for public keys by default, starting with a private key generated by `openssh` or by `openssl`.

```
$ openssl genrsa -out privkey.pem 2048
Generating RSA private key, 2048 bit long modulus
........................+++
.....................................................+++
e is 65537 (0x10001)
$ openssl rsa -in privkey.pem -pubout > pubkey.pub
writing RSA key
```

```
$ hexdump -C pubkey.pub
00000000 2d 2d 2d 2d 2d 42 45 47 49 4e 20 50 55 42 4c 49 |-----BEGIN PUBLI|
00000010 43 20 4b 45 59 2d 2d 2d 2d 2d 0a 4d 49 49 42 49 |C KEY-----.MIIBI|
00000020 6a 41 4e 42 67 6b 71 68 6b 69 47 39 77 30 42 41 |jANBgkqhkiG9w0BA|
..
```

Here, we used the flag `genrsa` with `openssl` to generate a private key, and then derived the public key from that. (The private key contains all the information needed for the public key).

According to the Python `rsa` module docs

`stuvel.eu/files/python-rsa-doc/reference.html#functions`

this is a `PKCS#1.5` PEM-encoded public key file from OpenSSL.

PEM-encoding should not be confused with the key type. PEM and binary format DER are alternatives. What they call PEM encoding is the default. This phrase refers to the fact that there is a header and a footer and base64-encoded data in between. However, different key formats may all be PEM-encoded.

As mentioned above these files can be recognized because they contain the phrase

`BEGIN PUBLIC KEY` rather than `BEGIN RSA PUBLIC KEY`.

We can also use the conversion utility on the private key file generated in part A by `ssh-keygen`.

```
$ openssl rsa -in kf -pubout > kf.pub
writing RSA key
$ hexdump -C kf.pub
00000000 2d 2d 2d 2d 2d 42 45 47 49 4e 20 50 55 42 4c 49 |-----BEGIN PUBLI|
00000010 43 20 4b 45 59 2d 2d 2d 2d 2d 0a 4d 49 49 42 49 |C KEY-----.MIIBI|
00000020 6a 41 4e 42 67 6b 71 68 6b 69 47 39 77 30 42 41 |jANBgkqhkiG9w0BA|
```

```
$ openssl rsa -outform PEM -in ./kf -pubout > kf.pub.pem
writing RSA key
$ diff kf.pub kf.pub.pem
$
```

This public key format is referred to in the `rsa` module as a "PKCS#1.5 PEM-encoded public key file from OpenSSL." I'm not sure of the difference between 1 and 1.5.

Format C: `RSA`

The private key from above is distinguished by the text `BEGIN RSA PRIVATE KEY` and I'm going to refer to it here as type C `RSA` for this reason, and because I am not totally sure what format it really is.

It is PKCS #8, according to

`www.cryptopp.com/wiki/Keys_and_Formats`

According to the `openssl` docs

`www.openssl.org/docs/apps/rsa.html`

this command should work to generate this format for a public key:

```
openssl rsa -in kf -RSAPublicKey_out
```

but it does not work for me.

Type C `RSA` is the default for the Python `rsa` module, but the function used to load it is `load_pkcs1`. This conflicts with information cited above.

`openssl` has a utility to convert a private key to (and from) PKCS #8 format:

```
openssl pkcs8 -nocrypt -in kf -topk8 -out kf.8 openssl pkcs8
-nocrypt -in kf.8
```

but it won't work on the private keys generated in part A or part B.

Format D: `binary`

The same utility can generate a DER format, which is a binary format:

```
$ openssl rsa -outform DER -in ./kf -pubout > kf.pub.der
writing RSA key
$ hexdump -C kf.pub.der
00000000 30 82 01 22 30 0d 06 09 2a 86 48 86 f7 0d 01 01 |0.."0...*.H.....|
00000010 01 05 00 03 82 01 0f 00 30 82 01 0a 02 82 01 01 |........0.......|
00000020 00 c3 cf 0b 10 38 f6 ba 7f a1 61 10 b3 ec 2b 53 |.....8....a...+S|
```

In addition to these four there is also an XML format.

Interconversion

To summarize the above, we have (A) Open SSH `ssh-rsa`, (B) X.509/PKCS#1 with "BEGIN PUBLIC KEY" `openssl`, (C) PKCS#8 with "BEGIN RSA PUBLIC KEY" `RSA`, and (D) binary format DER or `binary`.

Both B and C are PEM format, so this does not distinguish them.

Programs to generate keys include:

`ssh-keygen` which generates type A `ssh-rsa` public keys and type C `RSA` private keys. It is advertised to emit other types using the $-e$ flag, but this is yet another format, not as expected.

When we used this previously we modified the output by using the private key to generate type B.

```
openssl rsa -in kf -pubout > ./kf.pem
```

`openssl` generates a private key and then a public key like so:

```
openssl genrsa -out kf.pem 2048
```

These are equivalent, since PEM is the default:

```
openssl rsa -in ./kf -pubout
openssl rsa -outform PEM -in ./kf -pubout
```

`openssl` also obeys the flag DER to generate the binary type D.

These `openssl` keys are type B keys. Type C is what the Python `rsa` module expects as the default but it can also read other types. I am not sure how to generate type C using the other utilities.

`stuvel.eu/rsa` `stuvel.eu/files/python-rsa-doc/reference.html\#functions`

It will not load a type B key:

```
>>> import rsa
>>> import utils
>>> data = utils.load_data('kf.pub.pem')
>>> pbk = rsa.PublicKey.load_pkcs1(data)
..   ValueError:  No PEM start marker "-----BEGIN RSA PUBLIC KEY-----"
found
```

It will load the private key from part A:

```
>>> data = utils.load_data('kf')
>>> pk = rsa.PrivateKey.load_pkcs1(data)
>>> pk.e
65537
```

Try the type D `der` format from part D:

```
>>> data = utils.load_data('kf.pub.der')
>>> pbk = rsa.PublicKey.load_pkcs1_openssl_der(data)
>>> pbk.e
65537
```

It works. Try a different function with the type B `openssl` format

```
>>> data = utils.load_data('kf.pub.pem')
>>> pbk = rsa.PublicKey.load_pkcs1_openssl_pem(data)
>>> pbk.e
65537
```

This also works. So that's the secret. This is a special `openssl` format.

According to the docs

These files can be recognised in that they start with BEGIN PUBLIC KEY rather than BEGIN RSA PUBLIC KEY.

```
>>> data = pbk.save_pkcs1()
>>> data[:50]
u'-----BEGIN RSA PUBLIC KEY-----
nMIIBCgKCAQEAw88LEDj'
```

```
>>> fn = 'x.txt'
>>> FH = open(fn,'w')
>>> FH.write(data)
>>> FH.close()
```

```
>>>

$ hexdump -C x.txt
00000000 2d 2d 2d 2d 2d 42 45 47 49 4e 20 52 53 41 20 50 |-----BEGIN RSA
P|
00000010 55 42 4c 49 43 20 4b 45 59 2d 2d 2d 2d 2d 0a 4d |UBLIC KEY-----.M|
00000020 49 49 42 43 67 4b 43 41 51 45 41 77 38 38 4c 45 |IIBCgKCAQEAw88LE|
00000030 44 6a 32 75 6e 2b 68 59 52 43 7a 37 43 74 54 6d |Dj2un+hYRCz7CtTm|
00000040 4c 2b 34 39 66 46 61 42 4a 57 77 31 31 4f 76 48 |L+49fFaBJWw11OvH|
00000050 69 44 30 4d 39 41 6e 76 74 32 47 53 39 59 62 0a |iD0M9Anvt2GS9Yb.|
..

$ hexdump -C kf.pub.pem
00000000 2d 2d 2d 2d 2d 42 45 47 49 4e 20 50 55 42 4c 49 |-----BEGIN PUBLI|
00000010 43 20 4b 45 59 2d 2d 2d 2d 2d 0a 4d 49 49 42 49 |C KEY-----.MIIBI|
00000020 6a 41 4e 42 67 6b 71 68 6b 69 47 39 77 30 42 41 |jANBgkqhkiG9w0BA|
00000030 51 45 46 41 41 4f 43 41 51 38 41 4d 49 49 42 43 |QEFAAOCAQ8AMIIBC|
00000040 67 4b 43 41 51 45 41 77 38 38 4c 45 44 6a 32 75 |gKCAQEAw88LEDj2u|
00000050 6e 2b 68 59 52 43 7a 37 43 74 54 0a 6d 4c 2b 34 |n+hYRCz7CtT.mL+4|
00000060 39 66 46 61 42 4a 57 77 31 31 4f 76 48 69 44 30 |9fFaBJWw11OvHiD0|
..
```

We can also use the private key data to generate a public key

```
>>> pbk2 = rsa.PublicKey(pk.n,pk.e)
```

And one can generate keys using the module

```
>>> (pubkey, privkey) = rsa.newkeys(2048)
>>> pubkey.e
65537
>>> privkey.d
10854
```

Suppose we generate keys with very small parameters and then dissect what the rsa module gives us. That is for another time.