Crypto-new

te

Dec 19, 2019

CONTENTS:

CHAPTER ONE

FUNDAMENTALS

Contents:

1.1 Practical Passwords

Password strategy

These days it seems that every useful website or service requires a password. Ideally, passwords should be "strong." A strong password should require a very large number of attempts before an attacker could guess it successfully. `abc123` and `pw` are not strong passwords.

Check if you've been "pwned":

https://haveibeenpwned.com

To achieve strength, the best passwords would be a random string of bits. However, for readability, they are usually represented as strings of printable characters. Also, to mitigate the possibility that a password has been compromised, policy typically dictates that they change periodically.

Almost no one can remember a regularly changing set of random strings. One solution is to write the strings to a file and encrypt that, and then remember a single password that unlocks the file. The master password might be generated from a series of words with additional punctuation and unusual capitalization, something like:

`*I*amthe_greatesT#`

or

`*I*amthe_greatesT#a9b3c6`

I think that having a list of passwords encrypted and stored on disk is a good solution for most situations, e.g. my Amazon account or work email, where for the most part, any damage that might be done with a compromised credential can be reverted. Advantages include the ability to cut and paste, and cloud storage available to multiple devices. When combined with challenge-response via smartphone it is quite robust.

On the other hand, one could rely on physical possession of a password on a sheet of paper, entering it at the keyboard for each use. This is my approach for the most sensitive passwords, such as banking site passwords.

Security is ultimately a relative judgement. If the NSA is interested in you, it will be very difficult to keep your secrets.

Here is a simple Python script to generate passwords:

```python
import random
random.seed(153)
L1 = list('abcdefghijklmnopqrstuvwxyz')
L2 = list('0123456789')
L = [c + d for c in L1 for d in L2]
```

(continues on next page)

(continued from previous page)

```
N = 10
pL = [random.choice(L) for i in range(N)]
print ''.join(pL)
```

Test it:

```
> python pwgen.py
z8b9g7w1x2o7p5v8z3x0
>
```

To generate passwords which are actually random, comment out the second line.

Modify the script to accept the length as an argument on the command line, or just edit the value of *N*. If the password must be typed rather than copy-pasted, you could offset the groups so it's easier to read:

```
sep = ' '
print sep.join(pL)
```

```
>>> python pwgen.py
z8 b9 g7 w1 x2 o7 p5 v8 z3 x0
>>>
```

Knowledge-based authentication

In general, some kind of two-factor authentication is a good thing. One form of that is "Knowledge-based", where you login with username and password, and then the website challenges you with a question.

The classic one is, of course, what is your mother's maiden name? But there are plenty of others, as I'm sure you know.

A twist on this is when the server generates a question for you on its own. This happened for me with a Chase credit card, where they came up with a question (don't ask me how) and I didn't know the answer. Since they also had an old phone number, I was stuck.

New challenge questions can also happen by design. The site queries one of the credit agencies for your old addresses, presents a list, and then asks at which ones have you lived or *not* lived.

My take on this is that it is fine, with one proviso. The website *must allow you to set your own answers*, and they should allow anything. For example:

```
Q: Where was your maternal grandmother born?
A: x2z0j7v3m3k9b3
```

It's also helpful if they will let you do a reset.

My favorite:

```
Q: What is your mother's birthday (mmdd)
A: 1563
```

Of course, 15 is not a valid month. But that's what is great about it!

Just keep the Q&A pairs in a file. If it's for a banking site, write it down, or encrypt the file on disk. Copy-and-paste.

Password complexity

You've probably seen sites that require you to pick characters from certain sets, for example, including at least one digit, or one punctuation character.

These rules are misguided. The reason is simply that the number of permutations *P* generated by a randomization method depends on the size of character set *SZ* and the length of the password *n* in the following way:

$$P = SZ^n$$

That is, *n* is much more important than *SZ*. Taking logarithms

log P = n(log SZ)

We see the issue: *P* goes linearly with *SZ* but exponentially with *n*.

As an example, a character set of *SZ* equal to 32 (lowercase English alphabet plus the digits 1-6) gives a length 10 password set with the number of permutations equal to

$$P = 32^{10} = (2^5)^{10} = 2^{50}$$

If we increase the size of the character set to 64 (base64-encoding) we get

$$P = 64^{10} = (2^6)^{10} = 2^{60}$$

permutations, an increase of a factor of 2^{10}. Doubling the size of the character set doubles the number of permutations for each character.

On the other hand, retaining the smaller character set but increasing the length by just two gives exactly the same number of permutations.

$$P = 32^{12} = (2^5)^{12} = 2^{60}$$

We can overcome the deficiency of a small character set by a small increase in length. For that matter, we could decrease the character set to 8 tokens and increase the length from 10 to 20 and still have the same complexity.

$$P = 8^{16} = (2^3)^{20} = 2^{60}$$

Our primary goal in restricting the character set is to make it easy to enter passwords by hand. I could note that the letters and digits are on separate screens on my iPhone, so it would make sense to restrict the character set to just the 26 lowercase characters. On the other hand, I am so old-fashioned that I almost never enter passwords on my phone.

In the example given above, we have a set of elements containing pairs of characters with a letter followed by a digit, like `z5`, and the size of the set is 260. A password of length 20 total characters like `z8b9g7w1x2o7p5v8z3x0` has a complexity

```
>>> 260**10
141167095653376000000000L
```

This compares with a password made from single characters randomly chosen from the combined set of letters plus digits and of the same total length

```
>>> 36**20
13367494538843734067838845976576L
```

In the latter case, the positions are all independent and the number of permutations is greater by a factor of nearly 10 million. But we can easily make up for that by a modest increase in password length. Increasing the length to 13 pairs (26 total characters) gives a permutation space that is twice as large as 20 total characters from the 36 character set.

```
>>> 1.0 * 260**13 / 36**20
1.8561091354807857
```

The smaller the character set, the easier it is to type.

And password length is everything. That is why I prefer the approach shown above for typed passwords. If there is no need to type, something like this suffices from the command line:

```
> openssl rand 12 -base64
esDld/a+nVvLV5he
```

Perhaps even better

```
> openssl rand 12 -hex
ba46fc688334fd6fcb113e04
```

We use the utility `openssl` to generate 12 random bytes and then convert the result to base64 or hexadecimal.

autocomplete=”on”

As an aside, some websites use a form element that instructs the browser not to allow the OS to remember your passwords. It looks like this:

```
<input class="login" type="text" value=""
size="20"  autocomplete="off">
```

This is easy to turn off:

http://telliott99.blogspot.com/2010/12/turn-autocompleteon.html

This is not just annoying, it’s wrong. Requiring users to type in their passwords each time is a strong incentive to use weak passwords. That’s bad.

Of course there is an alternative risk, that the user will somehow “save” his password in an internet cafe, or forget to logoff when done. That is to me a lesser a problem, but if it bothers you, well, don’t turn off **autocomplete=”off”**.

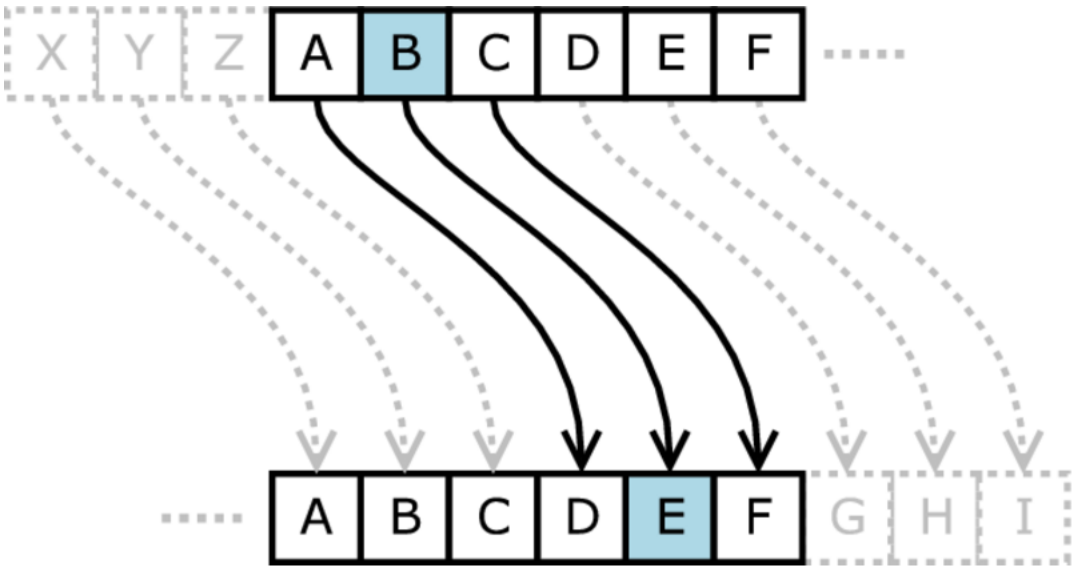
1.2 History

Caesar Cipher

Julius Caesar is said to have used a simple substitution cipher as an encryption system.

http://en.wikipedia.org/wiki/Caesar_cipher

> If he had anything confidential to say, he wrote it in cipher, that is, by so changing the order of the letters of the alphabet, that not a word could be made out. If anyone wishes to decipher these, and get at their meaning, he must substitute the fourth letter of the alphabet, namely D, for A, and so with the others. –Suetonius



Here is a quick and dirty Python script to do this:

`caesar.py`:

```
import sys

alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
o = int(sys.argv[1]) % 26
msg = ' '.join(sys.argv[2:])

vL = alpha[o:] + alpha[:o]
D = dict(zip(alpha,vL))

pL = list()
for m in msg:
    if m in D:
        pL.append(D[m])
    else:
        pL.append(' ')

print ''.join(pL)
```

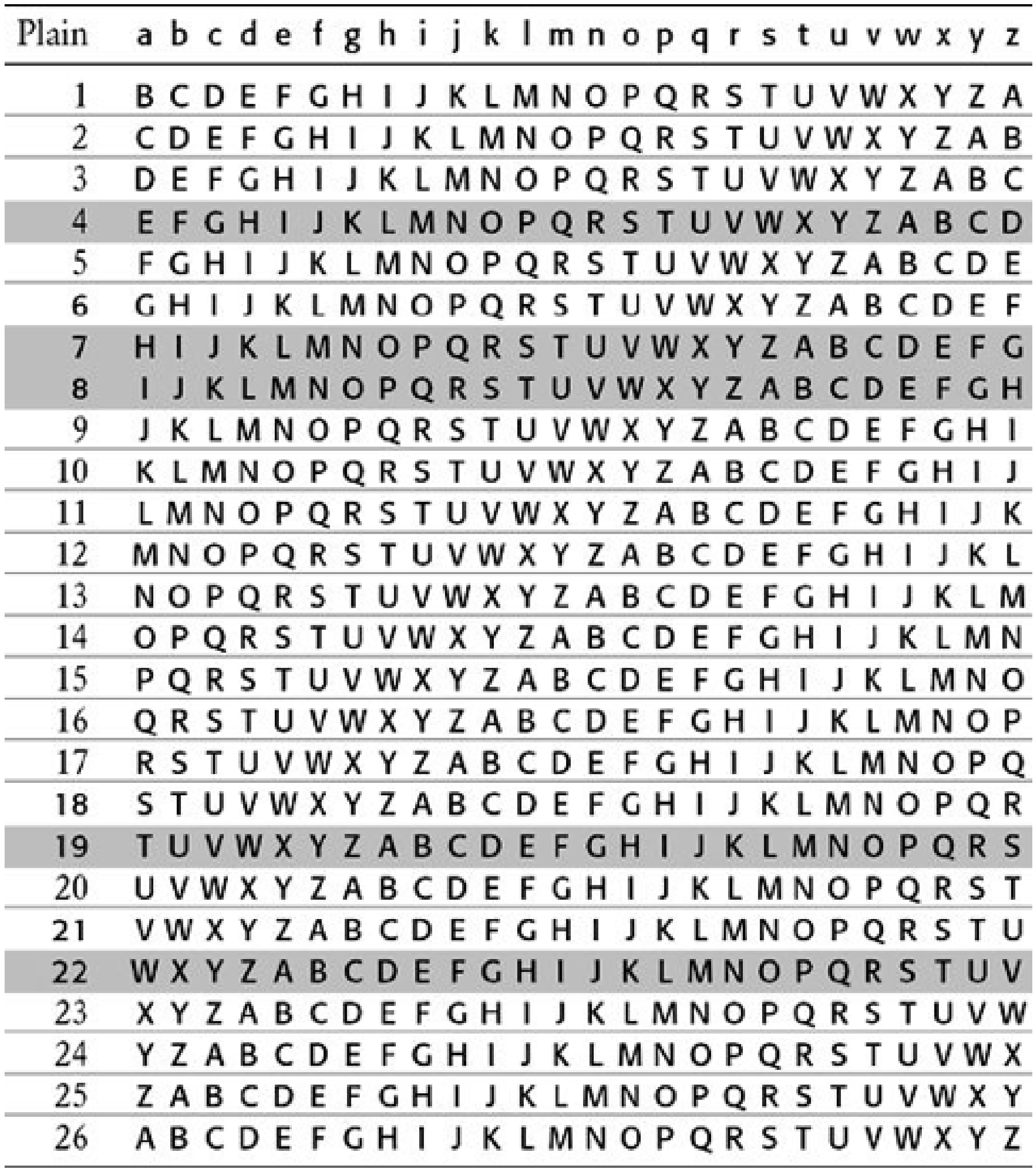
and here it is in action:

```
> python caesar.py 3 ABC
DEF
> python caesar.py 3 HI BRUTUS
KL EUXWXV
> python caesar.py -3 KL EUXWXV
HI BRUTUS
>
```

Simon Singh, in *The Code Book*

http://www.amazon.com/Code-Book-Science-Secrecy-Cryptography/dp/0385495323

writes about the Vigenere cipher. Here is a screenshot



Plain	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z
1	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
2	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
3	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
4	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
5	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
6	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
7	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
8	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
9	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
10	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
11	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
12	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
13	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
14	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
15	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
16	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
17	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
18	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
19	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
20	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
21	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
22	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
23	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
24	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
25	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y
26	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z

For each letter of a key, one makes a dictionary to encode the alphabet. For messages longer than the key, it repeats.

I wrote a Python script to do this. Here is the output

```
> python vigenere.py ATTACK
WABTGG
ATTACK
> python vigenere.py
A : ABCDEFGHIJKLMNOPQRSTUVWXYZ
B : BCDEFGHIJKLMNOPQRSTUVWXYZA
C : CDEFGHIJKLMNOPQRSTUVWXYZAB
```

(continues on next page)

(continued from previous page)

```
D : DEFGHIJKLMNOPQRSTUVWXYZABC
E : EFGHIJKLMNOPQRSTUVWXYZABCD
F : FGHIJKLMNOPQRSTUVWXYZABCDE
G : GHIJKLMNOPQRSTUVWXYZABCDEF
H : HIJKLMNOPQRSTUVWXYZABCDEFG
I : IJKLMNOPQRSTUVWXYZABCDEFGH
J : JKLMNOPQRSTUVWXYZABCDEFGHI
K : KLMNOPQRSTUVWXYZABCDEFGHIJ
L : LMNOPQRSTUVWXYZABCDEFGHIJK
M : MNOPQRSTUVWXYZABCDEFGHIJKL
N : NOPQRSTUVWXYZABCDEFGHIJKLM
O : OPQRSTUVWXYZABCDEFGHIJKLMN
P : PQRSTUVWXYZABCDEFGHIJKLMNO
Q : QRSTUVWXYZABCDEFGHIJKLMNOP
R : RSTUVWXYZABCDEFGHIJKLMNOPQ
S : STUVWXYZABCDEFGHIJKLMNOPQR
T : TUVWXYZABCDEFGHIJKLMNOPQRS
U : UVWXYZABCDEFGHIJKLMNOPQRST
V : VWXYZABCDEFGHIJKLMNOPQRSTU
W : WXYZABCDEFGHIJKLMNOPQRSTUV
X : XYZABCDEFGHIJKLMNOPQRSTUVW
Y : YZABCDEFGHIJKLMNOPQRSTUVWX
Z : ZABCDEFGHIJKLMNOPQRSTUVWXY
>
```

Invocation without a message shows the individual dictionaries. Of course, one could substitute any dictionary for these, for example, a different randomly scrambled one for each letter of the key. Nevertheless, even a scrambled system is easily broken by frequency analysis, given a long enough ciphertext. We will look at how this is done in a later section.

The fundamental point is that key repetition makes for “broken” cryptography.

`vigenere.py`:

```python
import sys

alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

D = dict()
for i,c in enumerate(alpha):
    values = alpha[i:] + alpha[:i]
    eD = dict(zip(alpha,values))
    D[c] = eD

def pretty_repr(D):
    kL = sorted(D.keys())
    k_str = ''.join(kL)
    vL = [D[k] for k in kL]
    v_str = ''.join(vL)
    return k_str,v_str

def vigenere_generator(key):
    L = [D[c] for c in key]
    i = 0
    while True:
        yield L[i]
        i += 1
```

(continues on next page)

(continued from previous page)

```
        if i == len(L):
            i = 0

def test():
    for c in alpha:
        eD = D[c]
        ks,vs = pretty_repr(eD)
        print c, ':', vs

def encrypt(msg,key):
    gen = vigenere_generator(key)
    rL = list()
    for m in msg:
        eD = gen.next()
        c = eD[m]
        rL.append(c)
    return ''.join(rL)

def decrypt(ctext,key):
    gen = vigenere_generator(key)
    rL = list()
    for c in ctext:
        eD = gen.next()
        for p in eD:
            if eD[p] == c:
                break
        rL.append(p)
    return ''.join(rL)

if len(sys.argv) == 1:
    test()
    sys.exit()

msg = sys.argv[1]
key = "WHITE"
ctext = encrypt(msg,key)
print ctext
ptext = decrypt(ctext,key)
print ptext
```

1.3 Encryption using XOR

XOR is symmetric

The XOR (exclusive or, xor) operation takes two bits and generates a third bit using the rules:

- 0 ^ 0 = 1 ^ 1 = 0
- 0 ^ 1 = 1 ^ 0 = 1

It can also be applied to streams of bytes like so:

- p = 0101

- k = 0011
- c = 0110

A fundamental result is that XOR is symmetric:

> p ^ k = c ^ k

If we take our message to be a string of ASCII characters like 'hello world', we can encrypt the string by computing the XOR of the bytes with a string of random bytes as the **key**.

The resulting ciphertext will consist of random bytes that can be de-ciphered if the key is known, by simply repeating the XOR operation.

In principle, if Alice and Bob both know the key but no one else does, and if the key is long enough (and never re-used), then their encrypted messages to each cannot be read by an eavesdropper, Eve.

Modern cryptography is about providing tools to make sure these two provisions are satisfied. Making a long enough key typically involves a stream cipher, whereas transmitting the key over an open network involves asymmetric (public/private key) cryptography.

Let's just do an example

```
> echo -n "hello world" > p.txt
```

We echo the text to `p.txt`, with the `-n` flag so as not to have a newline added at the end.

Find out how long a key we need and generate the necessary random bytes with `openssl`:

```
> wc -c p.txt
      11 p.txt
> openssl rand 11 > kf
```

One would have expected a command line utility to do the XOR operation, but I couldn't find one. So, to Python:

`xor_script.py`

```
FH = open('p.txt')
ptext = FH.read()
FH.close()
pL = [ord(v) for v in ptext]

FH = open('kf','rb')
key = FH.read()
FH.close()
kL = [ord(v) for v in key]

cL = [p ^ k for p,k in zip(pL,kL)]
c = bytearray(cL)

FH = open('c','wb')
FH.write(c)
FH.close()
```

In Python, the XOR operator is `^`, and it operates on decimals, which we obtain here using `ord`. We convert the result back to bytes using `bytearray` and write it to disk.

I did `hexdump` sequentially on `p.txt`, `kf` and `c` and re-formatted the results.

```
68 65 6c 6c 6f 20 77 6f 72 6c 64
de 62 db 79 9c 86 b5 bc c3 a0 09
b6 07 b7 15 f3 a6 c2 d3 b1 cc 6d
```

Now, unless you are made of metal, you won't obviously see that, for example, `6 ^ b = d`, but it does.

Recall that hexadecimal `b` is decimal `11 = 8 + 2 + 1` and `d` is decimal `13 = 8 + 4 + 1`, while `6 = 4 + 2` so:

```
6 = 0110
b = 1011
3 = 1101
```

Similar `a ^ c = 1010 ^ 1100 = 0110 = 6`, as seen in the leading position for the second byte from the end.

Anyway, it works. It is helpful to remember (or be able to count on your fingers to find) that

```
a = 10
b = 11
c = 12
d = 13
e = 14
f = 15
```

And then to factor for powers of 2 quickly.

```
13 = 8 + 4 + 1
```

And so on.

The result of this introductory material is devoted to solving the other problems, starting with key exchange.

1.4 Working with bytes in Python

In this section, I want to do a brief review of operations with data as bytes (rather than say, ASCII strings) in Python.

The quick take on this is that `int` is a central type, so to convert from hexadecimal to binary or back again, we go through an `int` as an intermediate

For example, the function `hex` converts an integer to its hexadecimal equivalent. The result has type `str`:

```
>>> h = hex(65)
>>> type(h)
<type 'str'>
>>> h
'0x41'
```

since, as we can calculate:

$$65 = 4 \times 16^1 + 1 \times 16^0$$

According to the docs, `hex` will take as input an integer of any size:

```
>>> hex(65537)
'0x10001'
>>> import math
>>> n = math.factorial(10000)
>>> h = hex(n)
>>> len(h)
29618
```

They're not kidding. Notice the result has `'0x'` only at the beginning of the string.

We convert the hexadecimal string back to an integer in a call to `int`, but we must specify the appropriate base (base 16):

```
>>> h = hex(65537)
>>> int(h,16)
65537
```

In the same way, we can get the binary representation of an integer using `bin`:

```
>>> bin(17)
'0b10001'
>>> bin(33)
'0b100001'
>>> b = bin(33)
>>> b
'0b100001'
>>> type(b)
<type 'str'>
```

As before with `hex`, the output from `bin` is a string.

The output loses leading one or more leading `0`. If you want to specify the padding, there are a couple of options. One easy way is to use `zfill`

```
>>> b
'0b01000001'
>>> b = bin(65)
>>> b
'0b1000001'
>>> b = '0b' + b[2:].zfill(8)
>>> b
'0b01000001'
```

Two other ways use fancy formatting

```
>>> '{:08b}'.format(65)
'01000001'
>>> format(65, '#010b')
'0b01000001'
>>>
```

but I think that simpler is better if it is sufficient.

Having generated a binary string, we can go back to the equivalent integer by specifying base 2 in a call to `int`, and then go on to hexadecimal from there if desired.

```
>>> b = bin(65)
>>> b
'0b1000001'
>>> int(b,2)
65
>>> hex(int(b,2))
'0x41'
```

Both of these examples so far are strings, representations of hexadecimal and binary numbers. They are not really binary data.

A quick search turns up Python classes `byte` and `bytearray`. (Much more is available in Python 3.x, but we are just using 2.7).

The first one is just an alias for `str`:

```
>>> L = [10, 32, 65]
>>> data = bytes(L)
>>> type(data)
<type 'str'>
>>> print data
[10, 32, 65]
>>> data
'[10, 32, 65]'
>>>
```

A `bytearray` can do more:

```
>>> L = [10, 32, 65]
>>> ba = bytearray(L)
>>> type(ba)
<type 'bytearray'>
>>> print ba
```

```
>>> for e in ba:
...     print e, type(e)
...
10 <type 'int'>
32 <type 'int'>
65 <type 'int'>
>>>
```

An iterator over a `bytearray` returns `int`. But if we print a `bytearray`, we get the string representation of the bytes, if possible, just as we did above.

http://stackoverflow.com/questions/7396849/convert-binary-to-ascii-and-vice-versa-python

Instead, an example of binary data might be obtained by entering the following:

```
>>> d = '\xff'
>>> d
'\xff'
>>> type(d)
<type 'str'>
>>> print d
?
>>> hex(255)
'0xff'
```

To enter binary data we use `\xff`. For a multibyte value each byte is preceded by `\x`.

(To do: explain the difference between `print` just evaluating the expression in the interpreter.)

The `print` function will use the printable character if there is one:

```
>>> print ('\x42')
B
>>> print ('\x0a')
```

```
>>>
```

It doesn’t format correctly here, but the newline gave us two empty lines. The interpreter starts output on a new blank line, then “prints” the newline (`LF`,’x0a’) by moving to a second blank line, then finally gives us the interpreter prompt `>>>` after moving again to a new line.

An example of multiple bytes:

```
>>> data = '\x9b\x3c'
>>> data
'\x9b<'
>>> type(data)
<type 'str'>
>>> print data
?<
```

We still seem to be dealing with strings. The hexadecimal value `3c` (`\x3c`) was printed as < in both tries.

```
>>> ord('<')
60
>>> hex(60)
'0x3c'
>>> type(hex(60))
<type 'str'>
>>> chr(60)
'<'
>>> print str(hex(60))
0x3c
```

The explanation is that < is the ASCII representation of the integer value 60, which is equal to the hexadecimal value `3c`. Since < is a printable character, that is what Python shows us. This has been a point of confusion for me in dealing with binary data in Python. Curiously, if we convert explicitly to string and then print, we don’t get the quotes.

```
>>> print hex(60)
0x3c
>>> hex(60)
'0x3c'
```

Here is a second example:

```
>>> import hashlib
>>> m = hashlib.md5()
>>> m.update('hello')
>>> d = m.digest()
>>> len(d)
16
>>> type(d)
<type 'str'>
>>> d
']A@*\xbcK*v\xb9q\x9d\x91\x10\x17\xc5\x92'
```

As before, if a printable character is available, Python will print that. Otherwise it prints `\x` plus the hex representation. In the above string, there are eight printable characters, each one byte (]A@*K*vq), plus another eight explicit bytes.

If you want the hexadecimal representation for all 16 of them, you could use `ord` to convert each value to an integer:

```
>>> L = [ord(y) for y in d]
>>> len(L)
16
>>> L[:4]
93
>>> L[:4]
[93, 65, 64, 42]
```

The last four are non-printing, and their integer values are > 128:

```
>>> L[-4:]
[16, 23, 197, 146]
>>> s = ''.join([hex(i) for i in L])
>>> s[:4]
'0x5d0x410x400x2a0xbc0x4b0x2a0x760xb90x710x9d0x910x100x170xc50x92'
```

If we write the data to disk

```
>>> FH = open('x.bin','wb')
>>> FH.write(d)
>>> FH.close()
```

If we examine this from the shell:

```
> hexdump -C x.bin
00000000  5d 41 40 2a bc 4b 2a 76  b9 71 9d 91 10 17 c5 92  |]A@*.K*v.q......|
00000010
```

And if we read it back in as binary data:

```
>>> FH = open('x.bin','rb')
>>> data = FH.read()
>>> data
']A@*\xbcK*v\xb9q\x9d\x91\x10\x17\xc5\x92'
>>> data == d
True
```

We can generate binary data from integers using the function `ord`. As an example, the last byte of the data above

```
>>> c = chr(146)
>>> type(c)
<type 'str'>
>>> c
'\x92'
```

This only works one byte at a time:

```
>>> chr(256)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: chr() arg not in range(256)
```

In summary, integers seem to be the unifying type. From integers we can go to hexadecimal (`hex`), 0s and *s (''bin'*) or binary data (`chr`). The complementary functions which return us to integers are `int` (with the appropriate base), or in the last case, `ord`.

We can use the `struct` module to look at multibyte data.

```
>>> L = [65,146]
>>> d = ''.join([chr(i) for i in L])
>>> d
'A\x92'
>>> len(d)
2
```

The last result should not be a surprise any more. If we want to go through the bytes one at a time, the reasonable way is to use `ord`. But another way would be:

```
>>> from struct import unpack
>>> unpack('B',d[0])
(65,)
>>> unpack('B',d[1])
(146,)
```

The advantage of this approach is we can get an integer (encoded in binary) in one step:

```
>>> h = '\x01\x00\x00\x01'
>>> h
'\x01\x00\x00\x01'
>>> type(h)
<type 'str'>
>>> unpack('I',h)
(16777217,)
```

We can check it:

```
>>> 256**3 + 1
16777217
```

Or just use `ord`:

```
>>> for c in h:
...     print ord(c)
...
1
0
0
1
```

1.5 Working with base64

In this section, I want to do a brief review of base64 encoding in Python.

The idea is to transform standard binary data, representable as integers in the range [0..255], to the base64 format, which uses a 64 character set. One reason to do that is the restricted set is safe for sending in URLs, while binary data is not.

64 characters require 6 bits. To do the encoding, we will transform a 3-byte (18 bit) sequence to 4 characters of 6 bits each.

The wikipedia example is a good place to start:

http://en.wikipedia.org/wiki/Base64

we should convert “Man” to “TWFu”.

As we said, it may be easiest to think of binary data as a sequence of integers in the range [0..255], though we can also think of it as a sequence of hexadecimal digit pairs in the range ['00'..'ff'], or even the ASCII equivalent, for those values that have a printable equivalent.

Thus 'M' has the value 77

```
>>> ord('M')
77
>>> ord('a')
97
>>> ord('n')
110
>>>
```

so 'Man' is really (77,97,110).

Now 77 is equal to 64 + 8 + 4 + 1 or '0b01001101'.

```
>>> bin(77)
'0b1001101'
>>>
```

Notice that Python strips the high value bits if they are equal to '0', and prepends '0b'. We can fix that with `zfill`:

```
>>> bin(77)[2:].zfill(8)
'01001101'
>>>
```

So

```
>>> def f(c):
...     return bin(c)[2:].zfill(8)
...
>>> ''.join([f(c) for c in [77,97,110]])
'010011010110000101101110'
>>> ' '.join([f(ord(c)) for c in 'Man'])
'01001101 01100001 01101110'
>>>
```

Now we take those as four 6-bit chunks:

> 010011 010110 000101 101110

And converting to decimal equivalent (base 10) we obtain: 19,22,5,46.

Now we just need to encode these digits. To get 64 characters, we use A..Z + a..z + 0..9 plus two more. The standard two additions are '+/', but sometimes others are used.

The 19th letter (counting from 0) is 'T', the 22nd is 'W', the 5th is 'F', and if you look up in the table you will find that 46 is 'u'.

So the base64 version of 'Man' is 'TWFu'

The only complication when encoding is if the binary data is not a multiple of 3. In that case, we add either one or two instances of the character '=' (decimal 61, binary '00111101).

With this background, it should be clear what the code below does. The basic routine converts a triplet of 3 integers [0..255] to a sequence of four integers [0..64].

We use that one to build up routines that take an ASCII string or a three-byte string in hexadecimal (6 digits).

Finally, I show routines from the `base64` module that do the encoding for us, starting from a string or hex data.

The code is really simple:

```
>>> import base64
>>> base64.b64encode('Man')
'TWFu'
>>> h = '0x4d616e'[2:].decode('hex')
>>> h
'Man'
>>> base64.b64encode(h)
'TWFu'
>>>
```

If the decode step above generates a non-printable letter, you will get something like this:

```
>>> 'ff'.decode('hex')
'\xff'
>>> '4d'.decode('hex')
'M'
```

One of the stumbling blocks that I had with this topic is that examples often take hex data (such as ‘0x4d616e’) and encode it as if it were ASCII. This is OK, but wasteful.

```
import random, base64

lc = 'abcdefghijklmnopqrstuvwxyz'
uc = lc.upper()
dg = '0123456789'
L = list(uc + lc + dg + '+/')
b64_dict = dict(zip(range(len(L)),L))

def binary_repr(n):
    # strip out the '0b', pad to 8 bits
    b = bin(n)[2:].zfill(8)
    return b

# takes   3 ints [0,255]
# returns 4 ints [0,63]
def group_bits_3to4(t):
    x,y,z = t
    x = binary_repr(x)
    y = binary_repr(y)
    z = binary_repr(z)

    a = x[:6]
    b = x[-2:] + y[:4]
    c = y[-4:] + z[:2]
    d = z[-6:]
    assert x + y + z == a + b + c + d

    a = int(a,2)
    b = int(b,2)
    c = int(c,2)
    d = int(d,2)
    return (a,b,c,d)

# t is a triplet of ints [0,255] (bytes)
def b64encode_ints(t):
    a,b,c,d = group_bits_3to4(t)
```

(continues on next page)

(continued from previous page)

```
    A = b64_dict[a]
    B = b64_dict[b]
    C = b64_dict[c]
    D = b64_dict[d]
    return A + B + C + D

# t is ASCII string, ignores extra values
def b64encode_ascii_str(s):
    while len(s) < 3:
        s += '='
    x = ord(s[0])
    y = ord(s[1])
    z = ord(s[2])
    return b64encode_ints((x,y,z))

# h is a triplet of bytes in hex
def b64encode_hex(h):
    if h[:2] == '0x':  h = h[2:]
    assert len(h) == 6
    x = int(h[:2],16)
    y = int(h[2:4],16)
    z = int(h[4:6],16)
    return b64encode_ints((x,y,z))

print b64encode_ascii_str('Man')
print b64encode_hex('0x4d616e')

print base64.b64encode('Man')
h = '0x4d616e'[2:].decode('hex')
print base64.b64encode(h)
```

When this script is run with `python script.py`, I obtain:

```
$ python script.py
TWFu
TWFu
TWFu
TWFu
$
```

1.6 Enigma machine

The Enigma encryption machine was used extensively by the German military during World War II

http://en.wikipedia.org/wiki/Enigma_machine

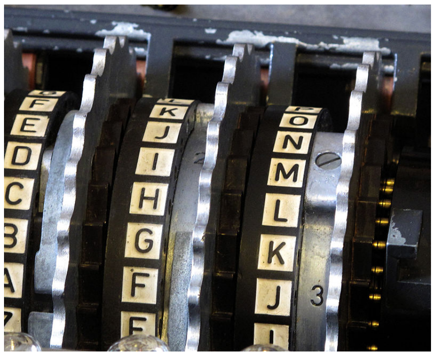
employs a series of *rotors*. A rotor is an electro-mechanical device that maps one ordering of the letters of the alphabet to another ordering.

As a simple example, we could imagine reversing the alphabet:

```
fwd  ABCDEFGHIJKLMNOPQRSTUVWXYZ
rev  ZYXWVUTSRQPONMLKJIHGFEDCBA
```

To encode a letter, find it in the alphabet on the first line, and then read off the encoding directly below.

Each version of Enigma was provided with a small number of rotors (usually 3) with fixed wiring, but the mapping could be changed by a process of *rotation* of an outside ring.



For example, the Enigma I rotors included

http://en.wikipedia.org/wiki/Enigma_rotor_details

three rotors with these mappings in the base state:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
EKMFLGDQVZNTOWYHXUSPAIBRCJ - I
AJDKSIRUXBLHWTMCQGZNPYFVOE - II
BDFHJLCPRTXVZNYEIWGAKMUSQO - III
```

As each letter from the plaintext is sequentially encoded, one or more of the rotors is rotated. We can model this like so:

step 1

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
EKMFLGDQVZNTOWYHXUSPAIBRCJ - I
```

step 2 (rotor I advanced one position)

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
KMFLGDQVZNTOWYHXUSPAIBRCJE - I +1
```

The pair of letters `AA` would thus be encoded as `EK`. The second is coding is different because rotor I has been advanced for step 2.

More complexity is achieved by using a series of rotors. The collection of rotors is works just like an odometer which records mileage in an automobile (or decimal addition). One complete turn of the rotor in the tenths place then advances the unit miles rotor by 1 step, and then a full turn of the unit miles rotor advances the tens place by 1.

In the schematic shown in wikipedia, the right hand rotor is the one that takes the plaintext letter input, and it is the one that rotates the fastest.

In this scheme, after encryption, letters from the plaintext are encoded one-at-a-time, so they retain the same relative positions in the ciphertext (although they are not encoded independently, as indicated above).

All traffic encrypted for a single message would use a particular starting arrangement of the available rotors.

Furthermore, a "letter" moving through the arrangement is *reflected* and sent backward through the same set of three rotors in reverse orientation. Consider the arrangements given above

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
EKMFLGDQVZNTOWYHXUSPAIBRCJ - I

```

(continues on next page)

(continued from previous page)

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
AJDKSIRUXBLHWTMCQGZNPYFVOE - II

ABCDEFGHIJKLMNOPQRSTUVWXYZ
BDFHJLCPRTXVZNYEIWGAKMUSQO - III
```

An *A* becomes sequentially *E* (rotor I), then *E* becomes *S* (rotor II) and finally *S* becomes *G* (rotor III).

After rotation, the letter is reflected and sent back through in reverse order.

There is more to the process, as explained below, but even at this point I had some misconceptions about the process that were cleared up by this authoritative reference

http://www.codesandciphers.org.uk/enigma/example1.htm

main page:

http://www.codesandciphers.org.uk/enigma/enigma1.htm

The example given has the rotors in the sequence I-II-III, with the current entering from the right. So the order of use is actually III-II-I-reflector-I-II-III.

This is just a matter of orientation. More important are these two facts: first, there is *no rotation* during any individual cycle.

Secondly, the rotors have the property that decryption is exactly *the same process* as encryption, running an `encrypt` function on the ciphertext yields the plaintext. This works because the cipher is symmetric, so that the pair of letters (AU) is enciphered as *A* -> *U* and *U* -> *A*.

Here are the three rotors in series:

```
ABCDEF G HIJKLMNO P QRSTUVWXYZ
BDFHJL C PRTXVZNY E IWGAKMUSQO - III

AB C D E FGHIJKLMNOPQRSTUVWXYZ
AJ D K S IRUXBLHWTMCQGZNPYFVOE - II

ABC D EFGHIJKLMNOPQR S TUVWXYZ
EKM F LGDQVZNTOWYHXU S PAIBRCJ - I
```

the standard reflector is

```
ABCDE F GHIJKLMNOPQR S TUVWXYZ
YRUHQ S LDPXNGOKMIEB F ZCWVJAT - R
```

The example given is

```
G -> C -> D -> F -> S
```

After that we must *invert the substitutions*. Here we have done that for each of the 3 rotors.

```
I
ABCDE F GHIJKLMNOPQR S TUVWXYZ
UWYGA D FPVZBECKMTHX S LRINQOJ - I r

II
ABC D EFGHIJKLMNOPQR S TUVWXYZ
AJP C ZWRLFBDKOTYUQG E NHXMIVS - II r
```

(continues on next page)

(continued from previous page)

```
III
AB C D E FGHIJKLMNOPQRSTUVWXYZ
TA G B P CSDQEUFVNZHYIXJWLRKOM - III r
```

Now, starting with *S* from the reflector we end up with *P*; overall, we have *G -> P*. Using the simulator below, we will show that *P -> G* as well. As mentioned, the operation of the rotors (in this combination and without "rotation") gives 13 pairs of letters which are simply exchanged:

```
(AU)(BE)(CJ)(DO)(FT)(GP)(HZ)(IW)(KN)(LS)(MR)(QV)(XY)
```

In actual operation, for the second letter one or more rotors have rotated according to the odometer model.

A last layer of encryption is provided by the "plugboard" The plugboard swaps pairs of letters, for example, *A* might become *T* and at the same time *T* becomes *A*.

Not all letters were switched in the plugboard, but most are. Normally, ten pairs were used. The plugboard settings would be changed each day.

In the language of linear algebra, we might express the algorithm as:

$$E = PRMLUL^{-1}M^{-1}R^{-1}P^{-1}$$

remembering of course, that the rotors `I`, `II`, `III` may advance with each letter (and one of them always does) as described above.

I have written a simulator in Python. Normally, I don't do much object-oriented programming, but this is a case where rotor "objects" are an obvious fit. Also, the plugboard and reflector can be modeled with the same rotor object, we just never rotate them. I draw the line at class inheritance. :)

At the end below is the listing for the simulator and its utilities (first version, no rotation). But before that, here is the output of a run with the rotors described above but no plugboard:

```
> python enigma.py
A -> B -> J -> Z -> T -> L -> K -> U
B -> D -> K -> N -> K -> B -> J -> E
C -> F -> I -> V -> W -> N -> T -> J
D -> H -> U -> A -> Y -> O -> Y -> O
E -> J -> B -> K -> N -> K -> D -> B
F -> L -> H -> Q -> E -> A -> A -> T
G -> C -> D -> F -> S -> S -> E -> P
H -> P -> C -> M -> O -> M -> O -> Z
I -> R -> G -> D -> H -> P -> U -> W
J -> T -> N -> W -> V -> I -> F -> C
K -> X -> V -> I -> P -> T -> N -> N
L -> V -> Y -> C -> U -> R -> G -> S
M -> Z -> E -> L -> G -> F -> W -> R
N -> N -> T -> P -> I -> V -> X -> K
O -> Y -> O -> Y -> A -> U -> H -> D
P -> E -> S -> S -> F -> D -> C -> G
Q -> I -> X -> R -> B -> W -> M -> V
R -> W -> F -> G -> L -> E -> Z -> M
S -> G -> R -> U -> C -> Y -> V -> L
T -> A -> A -> E -> Q -> H -> L -> F
U -> K -> L -> T -> Z -> J -> B -> A
V -> M -> W -> B -> R -> X -> I -> Q
W -> U -> P -> H -> D -> G -> R -> I
X -> S -> Z -> J -> X -> Q -> Q -> Y
Y -> Q -> Q -> X -> J -> Z -> S -> X
```

(continues on next page)

(continued from previous page)

```
Z -> O -> M -> O -> M -> C -> P -> H
>
```

Notice that we have generated all the pairs described above:

```
(AU)(BE)(CJ)(DO)(FT)(GP)(HZ)(IW)(KN)(LS)(MR)(QV)(XY)
```

and here a second run utilizing this plugboard

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
OWUREZGPYLKJMQAHNDVTCSBXIF
```

```
> python enigma.py
A -> O -> Y -> O -> Y -> A -> U -> H -> D -> R
B -> W -> U -> P -> H -> D -> G -> R -> I -> Y
C -> U -> K -> L -> T -> Z -> J -> B -> A -> O
D -> R -> W -> F -> G -> L -> E -> Z -> M -> M
E -> E -> J -> B -> K -> N -> K -> D -> B -> W
F -> Z -> O -> M -> O -> M -> C -> P -> H -> P
G -> G -> C -> D -> F -> S -> S -> E -> P -> H
H -> P -> E -> S -> S -> F -> D -> C -> G -> G
I -> Y -> Q -> Q -> X -> J -> Z -> S -> X -> X
J -> L -> V -> Y -> C -> U -> R -> G -> S -> V
K -> K -> X -> V -> I -> P -> T -> N -> N -> Q
L -> J -> T -> N -> W -> V -> I -> F -> C -> U
M -> M -> Z -> E -> L -> G -> F -> W -> R -> D
N -> Q -> I -> X -> R -> B -> W -> M -> V -> S
O -> A -> B -> J -> Z -> T -> L -> K -> U -> C
P -> H -> P -> C -> M -> O -> M -> O -> Z -> F
Q -> N -> N -> T -> P -> I -> V -> X -> K -> K
R -> D -> H -> U -> A -> Y -> O -> Y -> O -> A
S -> V -> M -> W -> B -> R -> X -> I -> Q -> N
T -> T -> A -> A -> E -> Q -> H -> L -> F -> Z
U -> C -> F -> I -> V -> W -> N -> T -> J -> L
V -> S -> G -> R -> U -> C -> Y -> V -> L -> J
W -> B -> D -> K -> N -> K -> B -> J -> E -> E
X -> X -> S -> Z -> J -> X -> Q -> Q -> Y -> I
Y -> I -> R -> G -> D -> H -> P -> U -> W -> B
Z -> F -> L -> H -> Q -> E -> A -> A -> T -> T
>
```

Notice that with inclusion of the plugboard, the output has changed but it still has the reflective symmetry property: encoding is reversible: A -> R and R -> A. This is a desired property of the design, to decrypt a message one starts at the same rotor settings as for encryption, and moves forward in parallel, working on the ciphertext.

`enigma_util1.py`:

```python
import random
random.seed(1337)

alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

def make_pb_seq():
    L = range(len(alpha))
    pL = list(alpha)
    random.shuffle(L)
```

(continues on next page)

(continued from previous page)

```
    for i in range(10):
        j = L.pop()
        k = L.pop()
        pL[j],pL[k] = pL[k],pL[j]
    return ''.join(pL)

def reverse_dict(D):
    rD = dict()
    for k in D:
        v = D[k]
        rD[v] = k
    return rD

def dict_to_string(D):
    L = [D[k] for k in alpha]
    return ''.join(L)

if __name__ == "__main__":
    seqP = make_pb_seq()
    print seqP
```

enigma1.py:

```
import sys, random
import enigma_util as ut

alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
seqA =  'EKMFLGDQVZNTOWYHXUSPAIBRCJ'
seqB =  'AJDKSIRUXBLHWTMCQGZNPYFVOE'
seqC =  'BDFHJLCPRTXVZNYEIWGAKMUSQO'
seqR =  'YRUHQSLDPXNGOKMIEBFZCWVJAT'

      # 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
      # 'OWUREZGPYLKJMQAHNDVTCSBXIF'
seqP =  ut.make_pb_seq()

class sub_cipher:
    def __init__(self,name,seq):
        self.seq = seq
        self.fD = dict(zip(alpha,seq))
        self.rD = ut.reverse_dict(self.fD)

    def printable_repr(self,d='fwd'):
        if d == 'fwd':
            D = self.fD
        else:
            D = self.rD
        s = self.name + ': '
        s += ut.dict_to_string(D)
        return s

pb = sub_cipher('pb', seqP)
r1 = sub_cipher('r1', seqA)
r2 = sub_cipher('r2', seqB)
r3 = sub_cipher('r3', seqC)
re = sub_cipher('re', seqR)
```

(continues on next page)

(continued from previous page)

```
# order is pb.III.II.I.refl.I.II.III.pb

def do_sub(c,pb,rgroup,re):
    rL = [c]
    # fD is forward
    c = pb.fD[c];  rL.append(c)
    for g in rgroup:
        c = g.fD[c];  rL.append(c)
    # now reflect
    c = re.fD[c];  rL.append(c)
    # rD is reverse
    for g in rgroup[::-1]:
        c = g.rD[c];  rL.append(c)
    c = pb.rD[c];  rL.append(c)
    print ' -> '.join(rL)
    return c

rgroup = [r3,r2,r1]

def test():
    for m in alpha:
        do_sub(m,pb,rgroup,re)

test()
```

1.7 Enigma with rotation

This section contains a modified version of the simulator that incorporates rotation (of the rightmost-rotor only). Here is a run:

```
> python enigma.py ATTACKATDAWN
input:  ATTACKATDAWN
ctext:  RFALTWCQFHYT
ptext:  ATTACKATDAWN
> python enigma.py ATTACKYOUIDIOTSORIWILLBESOMADATTACK
input:  ATTACKYOUIDIOTSORIWILLBESOMADATTACK
ctext:  RFALTWXVZNCDZUADHHQTCGHPPYDZRLCVCXR
ptext:  ATTACKYOUIDIOTSORIWILLBESOMADATTACK
>
```

And here is the code. The modified `enigma_util.py`:

```
import random
random.seed(1337)

alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
rev =   'ZYXWVUTSRQPONMLKJIHGFEDCBA'

def make_pb_seq():
    L = range(len(alpha))
    pL = list(alpha)
    random.shuffle(L)
    for i in range(10):
        j = L.pop()
```

(continues on next page)

(continued from previous page)

```
        k = L.pop()
        pL[j],pL[k] = pL[k],pL[j]
    return ''.join(pL)

def reverse_dict(D):
    rD = dict()
    for k in D:
        v = D[k]
        rD[v] = k
    return rD

def dict_to_string(D):
    L = [D[k] for k in alpha]
    return ''.join(L)

def rotate_dict(D,n):
    # values are *unique*
    vL = [D[k] for k in alpha]
    vL = vL[n:] + vL[:n]
    rD = dict()
    for k,v in zip(alpha,vL):
        rD[k] = v
    return rD

if __name__ == "__main__":
    seqP = make_pb_seq()
    print seqP
    D = dict(zip(alpha,rev))
    rotate_dict(D,1)
    for k in sorted(D.keys()):
        print k, D[k]
```

enigma.py:

```
import sys, random
import enigma_util as ut

alpha = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

# rotors
seqA =  'EKMFLGDQVZNTOWYHXUSPAIBRCJ'
seqB =  'AJDKSIRUXBLHWTMCQGZNPYFVOE'
seqC =  'BDFHJLCPRTXVZNYEIWGAKMUSQO'

# reflector
seqR =  'YRUHQSLDPXNGOKMIEBFZCWVJAT'

# plugboard
seqP =  ut.make_pb_seq()
      # 'OWUREZGPYLKJMQAHNDVTCSBXIF'
      # 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

class sub_cipher:
    def __init__(self,name,seq):
        self.name = name
        self.seq = seq
```

(continues on next page)

(continued from previous page)

```
        self.reset()

    def reset(self):
        self.fD = dict(zip(alpha,self.seq))
        self.rD = ut.reverse_dict(self.fD)
        self.i = 0

    def printable_repr(self,d='fwd'):
        if d == 'fwd':
            D = self.fD
        else:
            D = self.rD
        s = self.name + ': '
        s += ut.dict_to_string(D)
        return s

    def rotate(self,n=1):
        D = ut.rotate_dict(self.fD,n)
        self.fD = D
        self.rD = ut.reverse_dict(self.fD)
        self.i += n
        if self.i > 25:
            self.i -= 25

def init_components():
    pb = sub_cipher('pb', seqP)
    # order is pb.III.II.I.refl.I.II.III.pb
    # physical order is refl.I.II.III.pb <- input
    left = sub_cipher('r1', seqA)
    mid = sub_cipher('r2', seqB)
    right = sub_cipher('r3', seqC)
    grp = [right,mid,left]
    re = sub_cipher('re', seqR)
    return pb,grp,re

def do_sub(c,pb,rgroup,re,v=False):
    rL = [c]
    # fD is forward
    c = pb.fD[c];  rL.append(c)
    for g in rgroup:
        c = g.fD[c];  rL.append(c)
    # now reflect
    c = re.fD[c];  rL.append(c)
    # rD is reverse
    for g in rgroup[::-1]:
        c = g.rD[c];  rL.append(c)
    c = pb.rD[c];  rL.append(c)
    if v:  print ' -> '.join(rL)
    return c

# doesn't work properly yet
def rotate_all(grp):
    # rotation in order
    # 0-th rotor turns fastest
    right,mid,left = grp
    MAX = len(alpha) - 1
    if right.i == MAX:
```

(continues on next page)

(continued from previous page)

```
        if mid.i == MAX:
            if left.i == MAX:
                left.rotate()
            mid.rotate()
        right.rotate()

def test():
    for m in alpha:
        do_sub(m,pb,grp,re,v=True)

if __name__ == "__main__":
    #test()
    msg = sys.argv[1]
    print 'input: ', msg
    pb,grp,re = init_components()
    right,mid,left = grp

    ctext = list()
    for m in msg:
        c = do_sub(m,pb,grp,re)
        # rotate_all not working
        right.rotate()
        ctext.append(c)
    print 'ctext: ', ''.join(ctext)

    for g in grp:
        g.reset()

    ptext = list()
    for c in ctext:
        p = do_sub(c,pb,grp,re)
        right.rotate()
        ptext.append(p)
    print 'ptext: ', ''.join(ptext)
```

CHAPTER

TWO

RSA THEORY

Contents:

2.1 Messages and numbers

Before thinking about encryption, we need to work out just what a message is.

Let's write a message in English, for example, "hello world". What we will do is to generate a number representing the message, designated as *m*.

First, write this text into a file on disk

```
echo -n 'hello world' > x.txt
hexdump -C x.txt
```

I am going to write code so that it is formatted such that it can just be copied and pasted into a bash shell (above) or the Python interpreter (elsewhere, below).

The result is

```
00000000  68 65 6c 6c 6f 20 77 6f  72 6c 64        |hello world|
0000000b
```

What is actually present on the disk is the sequence of bytes `68 65 6c ...`.

That is to say, where `68` is written above, what is really present on disk is the binary sequence `01101000`, formed as a collection of elements in memory that either do or do not have a voltage across them.

Binary `01101000` corresponds to decimal `104` as well as hexadecimal `68`.

```
>>> int('01101000',2)
104
>>> hex(104)
'0x68'
```

2.1.1 ASCII

As you know, ASCII-encoding is an assignment of many of the 256 possible values for a byte to different codes or values, including letters such as: `68 : h`.

Since the letters are encoded in lexicographical order, the value `65` is `65 - 68 = -3` 3 places before `h`, namely, `e`, and `6c` is `6c - 68 = 4` places after `h`, i.e. `l`.

So `68 65 6c 6c 6f` is `hello`.

We could use the ASCII-encoding to generate a number. Write `hello` as its decimal equivalent sequence: `104-101-108-108-111`.

To turn this into a single number:

```
m = 104*256^4 + 101*256^3 + 108*256^2 + 108*256 + 111
```

```
m = 448378203247
```

For a longer message, just use more powers of 256.

In the code sample below, we convert the message `hello world` into a binary number using the `ord` and `bin` functions. Because bin doesn't show leading zeroes, we add them back with `zfill`.

```
s = "hello world"
L = [bin(ord(c))[2:].zfill(8) for c in s]
b = "".join(L)
```

Take a look

```
>>> int(b,2)
126207244316550804821666916L
>>>
```

However, this approach is fairly wasteful since most bytes never appear in any message (especially if we just use lowercase letters).

A more compact encoding might be achieved like this:

```
lc = 'abcdefghijklmnopqrstuvwxyz'
D = dict(zip(lc,range(1,len(lc)+1)))
D[" "] = 0
```

The dictionary `D` assigns an integer for each lowercase letter or a space. Now do:

```
s = 'hello world'
m = 0
N = 27
for i,c in enumerate(s):
    m += D[c] * N**i
```

The number *m* is being formed from the integer values for each character of `hello world`. For example, suppose the message is simply `hel`. Since `h` is the eighth letter, `e` is the fifth letter, and `l` is letter number 12, the value would be

```
8 + (5 × 27) + (12 × 27^2) = 8891
```

The result for 'hello world' can be viewed as:

```
python script2.py
```

```
920321254041092
```

The number *is* our message. To read it, just reverse the process:

```
N = 27
m = 920321254041092
```

(continues on next page)

(continued from previous page)

```
rD = dict(zip(D.values(),D.keys()))
pL = list()
while m:
    pL.append(rD[m % N])
    m /= N

print(''.join(pL))
```

Output:

```
>>> hello world
```

While the above encoding could be viewed as a form of encryption, it is pretty weak.

2.2 Math of RSA keys

Overview

We're exploring public-key cryptography. One first generates a pair of keys called the public key and a corresponding private (i.e. secret) key.

These have a symmetry property, such that if either one is used to encrypt a message, then the other one can be used to decrypt the ciphertext and recover the plaintext.

Suppose that Alice generates a public/private key pair and then distributes the public key in such a way that Bob is certain that he really knows Alice's public key.

Bob can encrypt a message with Alice's public key and send it to Alice with the knowledge that only Alice can decrypt it using her private key.

Alternatively, Alice can encrypt a message with her private key and send it to Bob. Successful decryption with Alice's public key proves that the message is from Alice.

The process of encryption and decryption is computationally expensive, and only short messages should be exchanged by this method. Such a message would typically consist of yet another key which can be used to encode the main message by a different protocol.

Two primes

Key generation starts with two prime numbers, p and q.

For a first example, I obtained two such numbers from a list of primes (near the 50 millionth prime):

The prime numbers for real-world use are much larger than this.

Python code:

```
p = 961748941
q = 982451653
n = p*q
```

If you paste the lines above into the Python interpreter, you should be able to query the value of n like this:

```
>>> n
944871836856449473
>>> len(bin(n))
62
>>>
```

This means we can securely encode not more than about 62 bits.

A note on notation: standard mathematics uses these two symbols for multiplication and exponentiation:

$$a \times b$$

$$a^e$$

We'll be using Python a lot, so we'll use the Python notation. Multiplication is `a * b` and exponentiation is `a ** e`. The remainder from division, or modulus operator, is `a % b`.

Phi

We're going to find three other numbers in order to use this method, one is called *phi* or sometimes *phi(n)* since it depends on the numbers p and q. We wouldn't mind using the Greek letter phi, but it's not part of ASCII and the LaTeX to PDF typesetter doesn't support it.

The other two values are called the exponents. The *exponent of the private key*, d, depends on phi.

We also need to choose e, the *exponent of the public key*.

```
phi = (p-1)*(q-1)
```

```
>>> phi
944871834912248880
>>>
```

Now choose e such that:

$$1 < e < phi$$

with the requirement that e should be coprime to phi, (they should have no common factor other than 1).

While we could factor phi, the easiest way to do this is to pick a prime number like

$$e = 2^{16} + 1 = 65537$$

Note that 65537 is *10000001* in binary.

I also confirmed that e is on a list of primes:

http://primes.utm.edu/lists/small/10000.txt

To test tha phi and e are coprime, we show that the prime number e is not a factor of phi, i.e. that there is a remainder when dividing:

```
>>> phi
944871834912248880
>>> e = 2**16 + 1
>>> e
65537
>>> phi % e
42186
>>>
```

The public key is then (n, e).

Actually, in practice, there is little variation in e. It is typically the same value as used here, 65537. According to Laurens Van Houtven's *Crpto 101*:

https://www.crypto101.io

e is either 65537 or 3, and this is because there are very few binary `1`'s in these numbers (only two single digits `1`), and as a result the exponentiation which we will compute *a lot*

```
m**e
```

is much more efficient.

65537 in binary is `10000 0000 0000 0001`. Thus to obtain `m^e`, just left-shift *m* by 16 and add that to the original value of *m* that we started with.

The private key consists of *n* plus another number *d* which is computed from phi and so, as we said, requires knowledge of *p* and *q*.

That is why the process of breaking this method of encryption is described as being the same as the problem of finding two primes that can factor a large number: *n*, the product of the primes *p* and *q*. *n* is known from the public key.

Encryption

The encryption function we will use is

$$c = m^e mod\ n$$

```
m = 920321254041092
e = 65537
n = 944871836856449473
x = m**e
```

```
>>> len(str(x))
980692
>>> c = x % n
>>> c
448344912451359241L
>>>
```

The number *c* is our ciphertext. (The L on the end signifies a Python long, a type of number).

$x = m^e$ is a very large number! Its decimal representation has nearly one million digits.

It is much more efficient to do the mod operation at the same time as the exponentiation. The Python built-in function `pow` allows that as an option:

```
>>> pow(m,e,n)
448344912451359241L
>>>
```

Decryption

We still need one more number to decode the encrypted text. This number is called *d*, the *exponent of the private key*.

The private key is (*d*, *n*), although just the *d* part is actually secret. Finding *d* is the tricky part of the whole operation, but it only needs to be computed once for a given key pair.

d is called the modular multiplicative inverse of *e* (mod *(n)*).

What this means is that we want *d* such that

```
d × e = 1 (mod phi(n))
```

Substituting the known values for *e* and *phi(n)*

```
d × 65537 = 1 (mod 944871834912248880)
```

Without worrying about the details, the method for doing this is the extended Euclidean algorithm for greatest common divisor. Given two numbers *a* and *b*, the egcd gives us *x* and *y* such that

$$ax + my = gcd(a, m)$$

So if the gcd is equal to 1 (and *e* and *phi* have been chosen with this in mind), then

$$ax - 1 = (-y)m$$

that is,

$$ax = 1 \; (mod \; m)$$

Thus, *x* is the multiplicative inverse of *a*, mod *m*.

Here is an online calculator. It appears to give the wrong answer!

I found an implementation for computing *d* here. We'll talk about this more in a separate chapter.

```
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b \% a, a)
        return (g, x - (b // a) * y, y)
```

```
def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception("modular inverse does not exist")
    else:
        return x % m
```

It is easy to show that this code does work. I saved it in a file `mod.py`. Let's try it out:

```
from mod import modinv
e = 65537
phi = 9448718349122488880
d = modinv(e,phi)
```

Output:

```
>>> d
8578341116816273
>>> d*e % phi
1L
>>>
```

So, having found *d*, now we are finally ready to decrypt:

```
c = 4483449124513592441
n = 9448718368564494473
d = 8578341116816273
p = pow(c,d,n)
```

```
>>> p
920321254041092L
>>>
```

Recall

```
>>> m
920321254041092
```

We have successfully generated a key pair, and used it to encrypt and decrypt a simple message. Now we need to show that we can also encrypt with private key, and decrypt with public one:

```
m = 920321254041092
d = 8578341116816273
n = 9448718368564494730
c = pow(m,d,n)
```

```
>>> c
4610006608697544451L
e = 65537
p = pow(c,e,n)
>>> p
920321254041092L
```

We have again recovered our plaintext message: *p* is equal to *m*.

2.3 Euclidean algorithm (EA)

As a result of our interest in modular arithmetic, we take a look at the famous Euclidean algorithm, which yields the *gcd* (greatest common divisor) of two integers *a* and *b*.

The reason this is important is that factoring large numbers is very difficult.

Consider the calculation of the greatest common divisor (gcd) of

`819 = 3 x 3 x 7 x 13`

`462 = 2 x 3 x 7 x 11`

We see immediately that the gcd `(819,462) = 21`.

The problem with this method is that, as we said, there is no efficient algorithm to factor integers.

The **gcd** algorithm consists of the following steps:

- compute `r = a % b`
- if `r == 0:  return b`
- set `a = b, b = r`

Example

```
  a =   b * r +  m
421 = 111 x 3 + 88
111 =  88 x 1 + 23
 88 =  23 x 3 + 19
 23 =  19 x 1 +  4
 19 =   4 x 4 +  3
  4 =   3 x 1 +  1
  3 =   1 x 3 +  0
```

The last non-zero remainder is 1, which we return as `b` when `r = 0`. This is gcd `(421,111)`.

gcd = `1` means that these two numbers do not share any factors. Hint: 421 is on this list, although 111 is not.

Here are three very similar Python implementations. We require `b < a` and if necessary we can do this before invoking the algorithmic code:

```
if b > a:
    a,b = b,a
```

My favorite loop `while True`:

```
 gcd(a,b):
    while True:
        r = a % b
        if r == 0:
            return b
        a,b = b,r

print gcd(421,111)   # 1
print gcd(60,24)     # 12
print gcd(11838*2888, 99991987*2888) # 2888
```

Although I like it, the *while True* irritates some people, who might prefer:

```
def gcd(a,b):
    r = a % b
    while r != 0:
        a,b = b,r
        r = a % b
    return b
```

And here is a recursive version, which calls itself:

```
def gcd(a,b):
    r = a % b
    if r == 0:
        return b
    return gcd(b,r)
```

Whatever suits you.

2.3.1 Explanation

Consider two integers *a* and *b* and we compute *m* = *a* mod *b*.

One possibility is that *a* is evenly divided by *b*, then the result of the mod operation is zero, and *b* is the gcd *(a,b)*.

If not, then either *a* and *b* have at least one common divisor smaller than *b* or they do not share a common factor (we do not consider *1* as a factor).

The mod operation can be expressed as

```
m = a - nb
```

where *n* can be computed variously as the "floor" of *a/b* (the next smallest integer from the real number that is computed), or the integer *n* such that

```
nb < a
```

but

```
(n+1)b > a
```

(If there were an integer *n* so that *nb* = *a* exactly, that would correspond to the case of zero remainder).

So we suppose *a* and *b* have a common factor *f*. Then we can factor *f* from each term of the previous equation:

```
m = a - nb
m = f(a/f - nb/f)
```

By the hypothesis of a common factor, the terms *a/f* and *nb/f* are integers.

But then clearly *m* is also evenly divided by *f* (because *m/f* is equal to an integer) and

```
m/f = a/f - nb/f
```

The insight is that now we can just find *gcd(b,m)*, since *b* and *m* also have the common factor *f*, and all the same logic applies.

It is easy to show that the algorithm always terminates, but I leave that aside for now.

2.4 Extended EA

The extended Euclidean algorithm is essentially the same as the Euclidean algorithm to find the gcd `(a,b)`, the greatest common divisors of two integers `a` and `b`.

However, it takes advantage of information that is generated in running the standard Euclidean algorithm, but which is normally discarded, in order to compute multiplicative inverses in modular arithmetic.

I found a nice page about this topic here .

2.4.1 Multiplicative inverses

Consider the multiplication table for two fields, one a prime number and one not prime.

```
    0   1   2   3   4
0   0   0   0   0   0
1   0   1   2   3   4
2   0   2   4   1   3
3   0   3   1   4   2
4   0   4   3   2   1
```

Above is the table for `Z5`. For each line, we generate all five members of `Z5` by one multiplication. `1` and `4` are each their own multiplicative inverses, while `2` and `3` are also inverses.

On the other hand, for `Zn` where `n` is not prime say, `Z8`, we might see:

```
    0   1   2   3   4   5   6   7
0   0   0   0   0   0   0   0   0
1   0   1   2   3   4   5   6   7
2   0   2   4   6   0   2   4   6
3   0   3   6   1   4   7   2   5
```

(continues on next page)

(continued from previous page)

```
4   0   4   0   4   0   4   0   4
5   0   5   2   7   4   1   6   3
6   0   6   4   2   0   6   4   2
7   0   7   6   5   4   3   2   1
```

In this case, the numbers which are prime, `3, 5, 7`, generate all of the elements of `Z`. The others do not, and furthermore, they never generate `1`. Thus `4` has no multiplicative inverse mod `8`, while `7 x 7` equals `1` mod `8`.

2.4.2 Extended Euclidean algorithm

I want to demonstrate the EEA but first I'd like to find a pair of numbers for which the gcd is equal to `1` and yet takes a few steps, starting from smallish numbers.

Let's try `231 = 3 x 7 x 11`, and `130 = 2 x 5 x 13`. Run the Euclidean algorithm as described in the previous chapter. Here is the code again:

```
def gcd(a,b):
    r = a % b
    while r != 0:
        a,b = b,r
        r = a % b
    return b
```

which gives:

```
  a    b    r   q
231  130  101   1
130  101   29   1
101   29   14   3
 29   14    1   2
 14    1    0  14
```

That looks reasonable. The `gcd` is the value of `b` when `r == 0`, that is, `1`.

Now, rearrange the data to be equations of the form `r = a - qb`.

```
101 = 231 - (1)130
 29 = 130 - (1)101
 14 = 101 - (3)29
  1 =  29 - (2)14
  0 =  14 - (1)14
```

We will use this data to write

```
xa - yb = 1
```

2.4.3 Backward

There are two related methods: forward and backward. The backward method starts with the next to last equation:

```
1 = 29 + (-2)14
```

Now, substitute for `14` from the previous equation

```
1 = 29 + (-2)[101 - 3(29)]
1 = (-2)101 + (7)29
```

Next, substitute for `29`:

```
1 = -(2)101 + (7)[130 - (1)101]
1 = (7)130 + (-9)101
```

At each stage, we can confirm that the arithmetic is correct. The equality is maintained.

Always the larger number has two terms to be combined. And the signs stay with the terms: all terms with `(--)101` will be negative, for example.

Then finally, substitute for `101`

```
1 = (7)130 + (-9)[231 - (1)130]
1 = (-9)231 + 16(130)
```

We have expressed the gcd as a *linear combination* of `a` and `b`. The form is

gcd = 1 = xa + yb 1 = (-9)231 + 16(130) 16(130) = (9)231 + 1

Now, realize that if we do mod `231` on both sides we have:

```
16(130) mod 231 = 1
```

Thus, `16` and `130` are modular multiplicative inverses mod `231`. We can check this easily:

```
>>> 16 * 130 % 231
1
```

Here we used four equations and *b* ended up with a positive factor. An odd number of equations would give a negative factor `y` in `yb`. In that case we take take the modulus of `y` by adding `a` to it.

2.4.4 Forward

Here are the equations again for reference.

```
101 = 231 - (1)130
 29 = 130 - (1)101
 14 = 101 - (3)29
  1 =  29 - (2)14
  0 =  14 - (1)14
```

In thinking about this, forget about the fact that the algorithm is constantly switching `a` for `b` and `b` for `r`. Here, `a` and `b` retain their initial values.

Start with the first equation:

```
101 = 231 - (1)130
    = a - b
```

The next line is

```
29 = 130 - (1)101
   = b + (-1)(a - b)
   = (-1)a + 2b
```

And the next:

```
14 = 101 - (3)29
   = (a - b) + (-3) [(-1)a + (2)b]
   = 4a - 7b
```

Finally,

```
1 = 29 + (-2)14
  = (-1)a + 2b + (-2)[4a - 7b]
  = -9a + 16b
```

These are the same values for `x` and `y` that we had from the backwards method.

The challenge now is to convert these algorithms to Python code.

2.5 Extended EA in code

We want to find multiplicative inverses, and to do that we will use the extended Euclidean algorithm.

First, here is a brute-force method which manually checks the remainder for every `n` until it finds one.

If the inverse does not exist, this function will never return.

```
# requires a > b
def caveman(m,p):
    n = 2
    while m*n % p != 1:
        n += 1
    return n
```

The inverse exists if and only if the greatest common divisor is `1`. In the case of arithmetic mod `60`, the integer `6` does *not* have a multiplicative inverse. That's because

```
(6 * i) % 60
```

cannot be equal to `1`. For mod `60` only the following numbers have inverses (as shown):

```
 7 43
11 11
13 37
17 53
19 19
23 47
29 29
31 31
37 13
41 41
43 7
47 23
49 49
53 17
59 59
```

If you're thinking these are all primes, you're right. Another way to state the problem with `6` and `60` is that their gcd is not equal to `1`: they have the common factor `6` or prime factors `2` and `3`.

2.5.1 Extended EA recap

In the previous chapter, we went through two methods for using the information from a run of the EA to compute the multiplicative inverse of a number mod a, the larger number input to the EA algorithm.

I call these the “forward” and “backward” methods.

The backward method runs the EA to completion first, then processes the data in reverse order. I found it relatively easy to code this one.

2.5.2 Backward code

With data from a run for `a = 231` and `b = 130`:

```
101 = 231 - (1)130
 29 = 130 - (1)101
 14 = 101 - (3)29
  1 =  29 - (2)14
  0 =  14 - (1)14
```

Consider this step:

```
1 = 29 + (-2)14
1 = 29 + (-2)[101 - 3(29)]
1 = (-2)101 + (7)29
```

We need two variables, call them `x` and `y`.

The data from the previous round is `x = 1` (the coefficient of `29`), and `y = -2` (the coefficient of `14`).

For the update, we need the new value of `q`, the quotient in the equation `q(b) = (3)29`.

The new values of `x` and `y` are:

:: x = y y = x - yq

where `x` in the second equation is the old value. We need a temporary variable to hold this value.

Here is the code for the inner loop from my version:

```
for i in range(2,N):
    a,b,q,r = L.pop(0)
    tmp = x
    x = y
    y = tmp - q * y
    t = a,b,q,r,x,y
    pp(t)   # pretty print
```

For the first step, the equation is:

```
1 =  29 - (2)14
```

so the values we need for `x` and `y` are:

x = 1 y = -q

Here is a run with some numbers I picked out of thin air:

```
> python eea.py 333337 58498
a=16,b=3,q=5,r=1,x=1,y=-5
a=35,b=16,q=2,r=3,x=-5,y=11
a=86,b=35,q=2,r=16,x=11,y=-27
a=465,b=86,q=5,r=35,x=-27,y=146
a=1016,b=465,q=2,r=86,x=146,y=-319
a=5545,b=1016,q=5,r=465,x=-319,y=1741
a=17651,b=5545,q=3,r=1016,x=1741,y=-5542
a=40847,b=17651,q=2,r=5545,x=-5542,y=12825
a=58498,b=40847,q=1,r=17651,x=12825,y=-18367
a=333337,b=58498,q=5,r=40847,x=-18367,y=104660
multiplicative inverse of 58498 is 104660 mod 333337
>
```

Check it:

```
>>> 58498 * 104660 % 333337
1
>>>
```

I found some code [here](https://en.wikibooks.org/wiki/Algorithm_Implementation/Mathematics/Extended_Euclidean_algorithm)

```
# return (g, x, y) a*x + b*y = gcd(x, y)
def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, x, y = egcd(b % a, a)
        return (g, y - (b // a) * x, x)
```

What this recursive version does in effect is the backward algorithm. It recursively calls itself, then builds up the result through all the returns.

It's a bit confusing because the values of `b` and `a` are switched compared to my usage: it does `b % a`. `x` and `y` are also switched.

The lines

```
g, x, y = egcd(b % a, a)
return (g, y - (b // a) * x, x)
```

carry out the computation, assigning `x` for `y`, and `y - q*x` for `x`, in the `return` line.

2.5.3 Forward code

The run for `a = 231` and `b = 130` had this data:

```
101 = 231 - (1)130
 29 = 130 - (1)101
 14 = 101 - (3)29
  1 =  29 - (2)14
  0 =  14 - (1)14
```

and these steps:

```
101 = 231 - (1)130
    = a - b

29 = 130 - (1)101
   = b - (1)(a - b)
   = (-1)a + 2b

14 = 101 - (3)29
   = (a - b) - (3)[(-1)a + (2)b]
   = 4a - 7b

1 = 29 + (-2)14
  = (-1)a + 2b - (2)[4a - 7b]
  = -9a + 16b
```

After noodling around for a bit, here is the pseudocode I came up with. We need to retain values from two rounds back, stored in the variables `s` and `t`.

```
round -1
s = 0
t = 1

round 0
q = 1
x = 1
y = -q = -1

round 1
q = 1
tmp = x,y = 1,-1
x = s - qx = 0 - 1 = -1
y = t - qy = 1 - (1)-1 = 2
s, t = tmp = 1,-1

round 2
q = 3
tmp = x,y = -1,2
x = s - qx = 1 - 3(-1)  = 4
y = t - qy = -1 - (3)2 = -7
s, t = tmp = -1,2

round 3
q = 2
tmp = x,y = 4,-7
x = s - qx = -1 - 2(4)  = -9
y = t - qy = 2 - (2)(-7)  = 16
s, t = tmp = 4,-7

y is the inverse
```

This is the inner loop:

```
```

while r != 0: a,b = b,r q = a / b r = a % b tmp = x,y x = s - q*x y = t - q*y s,t = tmp print ‘x’,x,’y’,y,’s’,s,’t’,t

[eea_forward.py](scripts/eea_forward.py)

```
> python eea_forward.py 231 130
    x 1 y -1
    x -1 y 2 s 1 t -1
    x 4 y -7 s -1 t 2
    x -9 y 16 s 4 t -7
    x 130 y -231 s -9 t 16
    16 is the inverse of 130 mod 231
>
```

Check it

```
>>> 16 * 130 % 231
1
>>>
```

Here is code from the web to carry out the forward method. I haven't figured out how it works yet.

```
def eea(a,b):
    s, t = 1, 0
    u, v = 0, 1
    while b != 0:
        q = a / b
        a, b = b, a % b

        tmp = s, t
        s = u - (q * s)
        t = v - (q * t)
        (u,v) = tmp
    return u

print eea(53,10)
```

This is very similar to what I had, except that `u,v` plays the role of `s,t`, and `s,t` plays the role of `x,y`.

2.6 Derivation of Euler's totient

Euler's totient function is symbolized by `phi(n)`.

It gives the count of how many numbers in the set `1..n-1` have a gcd with `n` equal to `1` (thus, it includes `1`). For a prime number `p`, `phi((p) = p-1`.

We will sketch a proof to show that if `n` has a prime factorization

$$n = p_1 \cdot p_2 \ldots$$

then

$$phi(n) = n(p_1 - 1/p_1)(p_2 - 1/p_2) \ \ldots$$

I found a nice write-up of this here:

http://www.claysturner.com/dsp/totient.pdf

2.6.1 Example

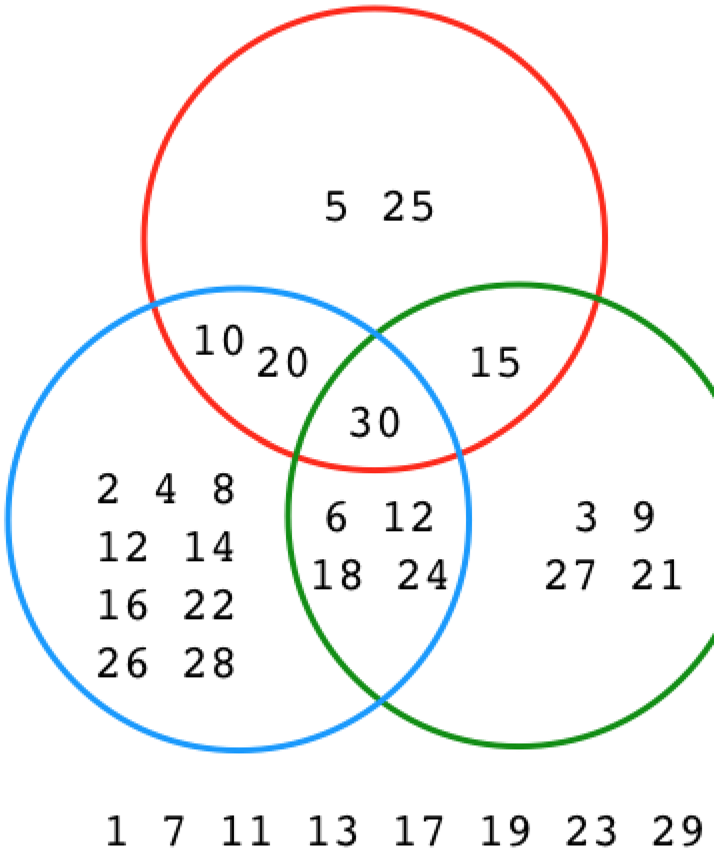
Consider `n = 30`.

If we run Euclid's algorithm on `1 < m < 30`:

```
>>> from gcd import gcd
>>> for i in range(2,30):
...     if gcd(30,i) == 1:
...         print(i)
...
7
11
13
17
19
23
29
>>>
```

The set we seek is `{ 1, 7, 11, 13, 17, 19, 23, 29 }`.

These are the numbers that are not included in the Venn circles:



The prime factors of `30` are `2`, `3` and `5`.

We can divide `1 <= m <= n` into several categories:

- 1
- `n`
- prime factor of `n`

plus

- gcd equal to a single prime factor: `2`, `3` or `5`
- gcd equal to any multiple of prime factors
- prime, not a factor of `n`

Now, what we are trying to count is the last line plus `1`, but what we will count are all the prime factors plus every number in this set

- gcd equal to a single prime factor: `2`, `3` or `5`

We must not double-count:

- gcd equal to any multiple of prime factors

plus `n`, and then subtract that from `30`.

Let

- A be the set of integers that 2 divides into evenly, plus `n`
- B be the set .. 3
- C be the set .. 5

We want to find the count or size of the set of all the numbers that are not in A, B or C (the numbers listed along the bottom of the figure).

This is

```
sizeof ~(A U B U C)
```

where ~ symbolizes the complement of the set, those elements not in the given set.

There is a famous theorem in set theory that says:

```
~(A U B U C) = ~A ^ ~B ^ ~C
```

so we write that we want

```
sizeof ~A ^ ~B ^ ~C
```

(where ^ is an ASCII symbol close enough to “intersection.”)

2.6.2 Probability

It is somewhat surprising, but we can calculate `~A`. Consider the set including `2` and all its multiples. The probability that a number `<= 30` is contained in the complement of that set is `1 - 1/2`, times `30`. Similarly, for `~B`, the probability is `1 - 1/3` times `30` and so on.

The total probability is the product of the individual probabilities.

And that total probability, times `n`, is equal to the count.

Thus

```
phi(30) = sizeof (~A ^ ~B ^ ~C)
= 30·(1 - 1/2)(1 - 1/3)(1 - 1/5)
```

And generalizing

```
phi(n) = n·(1 - 1/p_1)(1 - 1/p_2) \dots
```

which is what we needed to prove. This can be rewritten as

```
phi(n) = (p_1 - 1)(p_2 - 1) \dots
```

2.7 Why RSA works

phi I found a nice write-up of this here:

http://www.claysturner.com/dsp/totient.pdf

We'll attempt a high-level sketch of how/why RSA works here. At some point in the future, perhaps I'll fill in the details.

2.7.1 Euler's Totient

Euler's totient function is symbolized by `phi(n)`. It gives the count of how many numbers in the set

$$\{1, 2, 3 \dots, n-1\}$$

that is

$$a \in \mathbb{N} \mid a < n$$

share no common factors with `n` other than `1`, i.e. that have a gcd with `n` equal to 1. We then also include `1` in the count.

If `n` is prime, this is easy.

For a prime `p`, `phi(p) = p - 1`.

The only number that evenly divides `p` is `1` (and `1` always has a gcd of `1` with another integer), so the count of numbers less than `p` that have gcd = `1` is `p - 1`.

Otherwise, it gets harder. The fundamental theorem of arithmetic says that any number `n` has a *unique* prime factorization.

There is another theorem that says that if we have the prime factors of `n`:

$$n = p_1 \cdot p_2 \cdot p_3 \dots$$

then

$$phi(n) = phi(p_1) \cdot phi(p_2) \cdot phi(p_3) \dots$$

This is true not just for primes, but for any two coprime factors of `n`. This explains why we were able to write (in deriving an RSA key), that since `n = p * q`:

$$phi(n) = (p-1)(q-1)$$

So if

$$phi(n) = phi(p_1)\, phi(p_2)\, phi(p_3) \dots$$

We can multiply and divide by the prime product equal to `n` and obtain

$$phi(n) = \frac{p_1\ p_2 \ldots}{p_1\ p_2 \ldots}(p_1 - 1)\ (p_2 - 1)\ \ldots$$
$$phi(n) = n(p_1 - 1/p_1)(p_2 - 1/p_2)\ \ldots$$

In the write-up he sketches a proof of the last line above, and you can follow it backward to what we were given:

$$phi(n) = phi(p_1) \cdot phi(p_2) \cdot phi(p_3) \ldots$$

2.7.2 Fermat's Little Theorem

This theorem says that for any integer `a` and prime `p`:

$$a^p := a\ (mod\ p)$$

where **:=** means "defined to be equal".

We can rewrite this as

$$a^{p-1} = 1\ (mod\ p)$$

Euler proved this theorem (Fermat did not) and he extended it by showing that it is true, not just for `p` prime, but for any `b` coprime to a. Since we'll actually be using `n` (as defined before), let's stick with that:

$$a^{phi(n)} = 1\ (mod\ n)$$

2.7.3 Our goal

What we're looking for here is a function that has an inverse, where

$$(m^e)^d = (m^d)^e = m$$

Always, mod `n`.

Go back to Euler's extension of Fermat's little theorem (substituting `m` for `a`):

$$m^{phi(n)} = 1\ (mod\ n)$$

Raise to the power `k`:

$$m^{k \cdot phi(n)} = 1\ (mod\ n)$$

$$m^{k \cdot phi(n) + 1} = m\ (mod\ n)$$

So, we see that it will work to find

$$e \cdot d := k \cdot phi(n) + 1$$

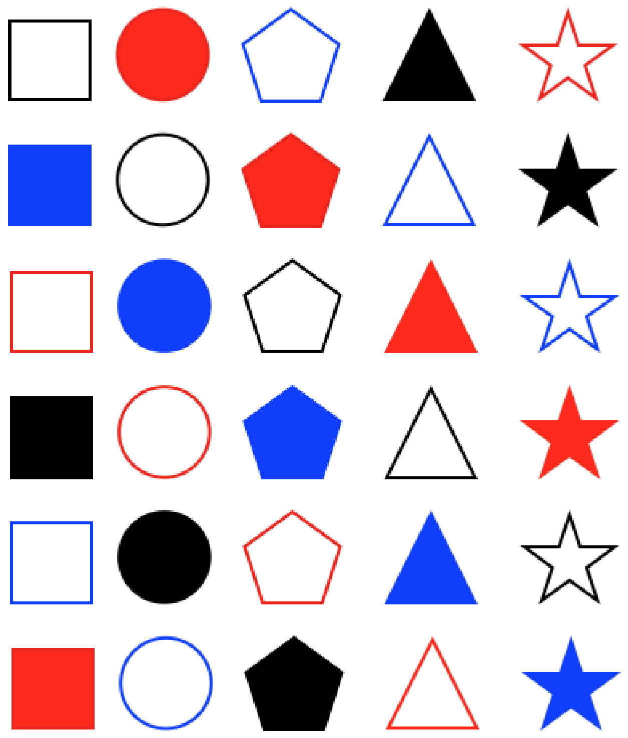
And thus

$$e \cdot d := 1\ mod\ phi(n)$$

And that's why it works.

2.8 CRT

The **Chinese** Remainder Theorem.



In the picture above, we have 5 distinct shapes, 3 different colors, and 2 states, either filled or empty.

As you can see, there are precisely 30 possible types.

For example, each shape can be one of three colors, filled or empty, so there are 6 possibilities for each of the five shapes.

It is easier to draw the picture if the shapes are laid out in a regular array, but once the assignments are made, the order is irrelevant.

This is simply a consequence of the fact that

$$5 \times 3 \times 2 = 30$$

Since there are 30 numbers in `[1..30]`, there is exactly one type for each number.

2.8.1 General statement

If N is composed of coprime factors

$$N = p \cdot q \cdot r \dots$$

and n is in the range `[1..N]`.

Suppose x has the set of remainders with those factors:

```
x = a mod p
x = b mod q
x = c mod r
```

Then this tuple `(a,b,c)` is unique to *x*.

2.8.2 Example

Suppose N = 60, so

```
30 = 2 x 3 x 5
```

Make a table of remainders:

```
    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
2:  1  0  1  0  1  0  1  0  1  0  1  0  1  0  1
3:  1  2  0  1  2  0  1  2  0  1  2  0  1  2  0
5:  1  2  3  4  0  1  2  3  4  0  1  2  3  4  0

   16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
2:  0  1  0  1  0  1  0  1  0  1  0  1  0  1  0
3:  1  2  0  1  2  0  1  2  0  1  2  0  1  2  0
5:  1  2  3  4  0  1  2  3  4  0  1  2  3  4  0
```

Any particular triplet of remainders is unique, say: (1,1,3) -> 13.

This is also true if the factors are not prime but simply co-prime:

```
    1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
5:  1  2  3  4  0  1  2  3  4  0  1  2  3  4  0
6:  1  2  3  4  5  0  1  2  3  4  5  0  1  2  3

   16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
5:  1  2  3  4  0  1  2  3  4  0  1  2  3  4  0
6:  4  5  0  1  2  3  4  5  0  1  2  3  4  5  0
```

2.9 RSA calculations

In the theory of RSA encryption and decryption, we choose two primes *p* and *q* and compute their product *n = pq*, as well as Euler's totient *phi(n) = (p-1)(q-1)*.

We pick a public exponent *e*, which is co-prime to *phi(n)*, and the usual choice is `65537 = 0x101 = 0b1 00000000 00000001`.

We must also find *d* such that it is the multiplicative modular inverse of *e*:

$$d = e^{-1} \, mod \, phi(n)$$

This is done using the extended Euclidean algorithm for gcd (greatest common divisor). The above equation is equivalent to:

$$e \cdot d = 1 \, mod \, phi(n)$$

With these definitions, it can be shown that

$$(m^e)^d = (m^d)^e = m \, mod \, n$$

This is the basis of asymmetric encryption and decryption.

It makes life easy if *e* has very few binary ones, which explains the choice we made above. However, *d* is determined from *e* and *phi(n)*, and computing

$$x^d \; mod \; n$$

is hard. (`x` is either the plaintext message or its encryption with the public key).

2.9.1 Shortcuts

The Chinese Remainder Theorem tells us that for each `x` in the range `[1..n-1]` there is a unique pair or tuple `(x mod p-1, x mod q-1).`

Some smart guy figured out that, not only is the result uniquely identified by that tuple, but it can be calculated from it.

So, just as with `phi(n)`, we calculate the multiplicative modular inverse of *e* with *(p-1)* and *(q-1)*. These are called

$$dP = e^{-1} \; mod \; (p-1)$$

$$dQ = e^{-1} \; mod \; (q-1)$$

Note that

$$dP \cdot e = 1 \; mod \; (p-1)$$

$$dQ \cdot e = 1 \; mod \; (e-1)$$

We also need the inverse of *q* with respect to *p*

$$qInv = q^{-1} \; mod \; p$$

To do the calculation, first find

$$x_1 = x^{dP} \; mod \; p$$

$$x_2 = x^{dQ} \; mod \; q$$

Then

$$x = x_2 + h \cdot q$$

where

$$h = (x_1 - x_2) \cdot qInv \; mod \; p$$

2.9.2 Example

It's nice to find a worked out example that tests for errors:

https://www.di-mgt.com.au/crt_rsa.html

Suppose

```
p = 137
q = 131
n = 17947
phi = 16680
e = 3
```

Compute *d* using `mod.py` (and then just check it):

```
d = 11787
```

So then

$$dP = e^{-1}\, mod\, (p - 1)$$

$$dP = e^{-1}\, mod\, (q - 1)$$

```
dP = 91
dQ = 87
```

and finally

$$qInv = q^{-1}\, mod\, p$$

```
qInv = 114
```

Having chosen a message with value `m = 513`, to encrypt:

$$c = (m^e)\, mod\, n$$

```
m = 513
c = 513^3 mod 17947 = 8363
```

2.9.3 Calculation

Decrypt by

```
8363^11787 mod 17947 = 513
```

The intermediate exponentiation result `8363^11787` has `46233` digits. It goes pretty quickly on my computer, but still..

The shortcut is as follows:

$$m_1 = c^{dP}\, modp$$

```
m1 = 8363^91 \ mod 137 = 102
```

Here, the exponentiation result has 357 digits.

$$m_2 = c^{dQ}\, modq$$

```
m2 = 8363^87 \ mod 131 = 120
```

Since `m2 > m1` the difference is negative, therefore add an extra `p`:

```
m1 - m2 = -18;  + 137 = 119
```

Multiply by `qInv`:

```
h = qInv.(m1-m2) mod p = 114 x 119 mod 137 = 3
```

So the result is

```
m = m2 + h.q = 120 + 3(131) = 513
```

It works!

CHAPTER

THREE

RSA PRACTICE

Contents:

3.1 RSA key formats

There are a number of different RSA key formats that you might run into.

- `OPENSSH PRIVATE` (OpenSSH format)
- `RSA PRIVATE/PUBLIC KEY` (PKCS1)
- `PRIVATE/PUBLIC KEY` (PKCS8, may be ENCRYPTED)
- `SSH2 PRIVATE/PUBLIC KEY` (RFC4716)

So far as I know, despite contrary OpenSSH usage, these are all technically PEM format (i.e. base64-encoded, with a first line BEGIN XXX), except the OpenSSH public key which starts ssh-rsa.

Even though it’s base64-encoded the first public key “ssh-rsa” is not PEM format because it doesn’t have the “BEGIN XXX” part.

And they can be distinguished by the first line.

3.1.1 OpenSSH

```
ssh-keygen -f kf
```

- —–BEGIN OPENSSH PRIVATE KEY—–
- ssh-rsa

The first is the first line of the private key, and the second is the public one. The private key is a propietary format.

According to the man pages for `keygen`:

These are “keys for use by SSH protocol version 2. . . For keys stored in the newer OpenSSH format, there is also a comment field in the key file that is only for convenience to the user to help identify the key.”

So this suggests there is an older and a newer OpenSSH format.

SSH v. 2 is described in RFC4253. Docs here: https://tools.ietf.org/html/rfc4253#section-6.6

Discussion: https://blog.oddbit.com/post/2011-05-08-converting-openssh-public-keys/

Regardless of format, the `openssl` utility will not read them.

```
openssl rsa -text -in kf
```

Output:

```
unable to load Private Key
```

3.1.2 PKCS1

```
ssh-keygen -f kf.pub -e -m pem
```

convert an OpenSSH public key to PKCS1 format:

- —–BEGIN RSA PUBLIC KEY—–

which is officially PKCS #1.

(Often called PKCS1, because of some technical restriction on the # symbol).

According to the internet, a PKCS1 public key looks like this:

```
RSAPublicKey ::= SEQUENCE {
    modulus           INTEGER,  -- n
    publicExponent    INTEGER   -- e
}
```

According to the man page for `ssh-keygen`, in the section on the flag -m key_format, there are three supported formats:

- RFC4716
- PKCS8 (PEM PKCS8 public key)
- PEM (PEM public key)

The last is PKCS1 which we saw above.

3.1.3 PKCS8

```
> ssh-keygen -f kf.pub -e -m pkcs8
-----BEGIN PUBLIC KEY-----
```

3.1.4 RFC4716

One description of the RFC4716 format is [here](https://tools.ietf.org/html/rfc4716). A distinguishing characteristic is

```
---- BEGIN SSH2 PUBLIC KEY ----
```

- says: BEGIN/END SSH2 PUBLIC KEY
- there are four dashes plus a space
- may be preceded by Comment: and Subject lines
- line length is 70 (required <= 72 bytes)
- line termination character can be native

- the key data is base64-encoded

3.2 Key generation

3.2.1 OpenSSH

Probably the most common way to generate RSA keys is to use the `ssh-keygen` utility:

```
ssh-keygen
```

Common flags include:

- `[-b bits]`
- `[-t dsa | ecdsa | ed25519 | rsa]`
- `[-N new_passphrase]`
- `[-f output_keyfile]`

The defaults are 2048 bits and RSA. It will prompt for a passphrase (which can be empty `""`) and for an output file path, which is suggested to be `.ssh/id_rsa` plus the same with `.pub`.

3.2.2 Examples

```
> ssh-keygen -N "" -f kf
Generating public/private rsa key pair.
Your identification has been saved in kf.
Your public key has been saved in kf.pub.
The key fingerprint is:
SHA256:Qom284Cw07VyzyAPJ3CZds+ssIFXF44XHN2HL/qBx/U telliott@newmini.local
The key's randomart image is:
+---[RSA 2048]----+
|      .+o . .     |
|   o +.+. o .    |
|o = B *    o     |
| O * X    . o    |
|+ @ O = S+ o .   |
| o % O .o +   E  |
|  . o +  o .     |
|          .      |
|                 |
+----[SHA256]-----+
```

kf

```
-----BEGIN OPENSSH PRIVATE KEY-----
..
```

kf.pub

```
ssh-rsa
..
```

This private key said to be OpenSSH format, but it is really already PKCS1.

3.2.3 Export

- `-e` This option will read a private or public OpenSSH key file and print to stdout a public key in one of the formats specified by the `-m` option. The default export format is "RFC4716".

```
---- BEGIN SSH2 PUBLIC KEY ----
```

To get PKCS1

```
ssh-keygen -f kf.pub -e -m pem
```

will convert the public key in file `key.pub` into PKCS1 format, and print it to standard out.

```
-----BEGIN RSA PUBLIC KEY-----
```

Using the `-e -m` flags:

```
ssh-keygen -f kf.pub -e -m pem > kf.pub.pem
```

saves the new key.

None of these is read by `openssl rsa`.

3.2.4 Import

- `-i` This option will read an unencrypted private (or public) key file in the format specified by the -m option and print an OpenSSH compatible private (or public) key to stdout.

Only this one worked so far:

```
ssh-keygen -i -f kf.pub.pem -m pem
```

RFC4716 is supposed to work.

3.3 OpenSSH fingerprints

3.3.1 Digests

```
> echo "hello" > x.txt
> md5 < x.txt
b1946ac92492d2347c6235b4d2611184
> openssl dgst -sha1 < x.txt
f572d396fae9206628714fb2ce00f72e94f2258f
> openssl dgst -sha256 < x.txt
5891b5b522d5df086d0ff0b110fbd9d21bb4fc7163af34d08286a2e846f6be03
```

```
> echo "hello" | md5
b1946ac92492d2347c6235b4d2611184
> echo "hallo" | md5
aee97cb3ad288ef0add6c6b5b5fae48a
```

Quick to compute, sensitive to subtle changes, impossible to craft a different set of data with the same hash (though this is changing, which is why SHA256 is now recommended).

3.3.2 Fingerprinting

The `ssh-keygen` utility can print a fingerprint for an OpenSSH key.

This is simply the hash of the binary data in the public key. It makes it easier to compare the data in the public key on a server with what you have on the client at home.

You can either input the public key, or the program will find the public key data embedded in the private key.

The command line flag is:

- `-l` fingerprint

as in

```
ssh-keygen -lf kf
```

which outputs something like:

```
2048 SHA256:Qom284Cw07VyzyAPJ3CZds+ssIFXF44XHN2HL/qBx/U telliott@newmini.local (RSA)
```

To get MD5 you just do

- `-E`

```
> ssh-keygen -l -f kf -E md5
2048 MD5:bd:9d:45:95:9f:a3:8f:88:20:93:40:76:66:9e:77:8b telliott@newmini.local (RSA)
```

3.3.3 Confirmation

I want to confirm that this is doing what I think it is. While it is simple enough to cut and paste the base64 data of the public key, as in:

```
echo ''
```

then paste the data in betweenca the quotes, and hit return:

```
AAAAB3Nz..lg3EH32T
```

A better way if you're going to do this a lot is to use `awk`.

```
awk '{ print $2 }' kf.pub
```

which gives the same thing. Then pipe that to

```
| base64 -d | openssl dgst -sha256 -binary | base64
```

That is, do

```
awk '{ print $2 }' kf.pub \
| base64 -d | openssl dgst -sha256 -binary | base64
```

with output:

```
Qom284Cw07VyzyAPJ3CZds+ssIFXF44XHN2HL/qBx/U=
```

or more simply:

```
> awk '{ print $2 }' kf.pub \
| base64 -d | md5
```

with output:

```
bd9d45959fa38f8820934076669e778b
```

And that's a match.

3.4 OpenSSL genrsa

3.4.1 genrsa

`openssl` has a utility for generating a private key, called `genrsa`:

```
openssl genrsa -out kf
```

```
-----BEGIN RSA PRIVATE KEY-----
..
```

Encryption is optional. The default public exponent is 65537.

The man page says `numbits` is how to specify size, but this doesn't work.

The output is PKCS1. There is no option for outputting a different format. However, there is another utility `openssl rsa` that can interconvert different formats.

Also, this method does not generate a public key file, you have to do that separately.

3.5 OpenSSL rsa

- derive a public key from a private key
- print a key's components in hex
- change from base64 (PEM) to binary (DER)

```
openssl rsa -in kf
```

Some flags:

- `-modulus` print the modulus (only)‘‘
- `-out file`
- `-outform der | net | pem`
- `-pubin` read a public key
- `-pubout` output a public key
- `-text` print the private key components in plain text

3.5.1 Examples

```
openssl rsa -in kf -pubout -out kf.pub
```

Output a public key PKCS8.

```
> openssl rsa -in kf -pubout
writing RSA key
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEA10R1rteU0JcuA74UEUoy
```

Print key components:

```
openssl rsa -in kf -text
```

Output

```
Private-Key: (2048 bit)
modulus:
00:d7:44:75:ae:d7:94:d0:97:2e:03:be:14:11:4a:
..
```

This utility will not read any of the OpenSSH keys, private or public, PKCS1 or not.

3.6 OpenSSL genpkey

`openssl` has another utility for generating keys called `genrsa`:

```
openssl genpkey -out kf2
```

"The genpkey command generates private keys. The use of this program is encouraged over the algorithm specific utilities because additional algorithm options can be used."

- `-algorithm alg` e.g. rsa
- `-out` file to write
- `-pkeyopt opt:value` like `rsa_keygen_bits:2048`
- `-text`

3.6.1 Examples

```
openssl genpkey -algorithm rsa > kf.private.pem
```

gives a PKCS8 formatted key in `kf.private.pem`.

```
-----BEGIN PRIVATE KEY-----
```

```
openssl genpkey -algorithm rsa -text
```

prints the whole thing including the formatted key, but also the modulus, etc. PKCS8

3.7 OpenSSL

As we said last time, standard key formats include:

- OPENSSH PRIVATE (OpenSSH format)
- RSA PRIVATE/PUBLIC KEY (PKCS1)
- PRIVATE/PUBLIC KEY (PKCS8, may be ENCRYPTED)
- SSH2 PRIVATE/PUBLIC KEY (RFC4716)

All of these are PEM-encoded save the OpenSSH Public key, which doesn't meet the standard first line "BEGIN XXX".

And, also we said that:

```
ssh-keygen -N "" -f kf
```

gives a new key in OpenSSH format.

`ssh-keygen` can convert to other formats with `-e -m`, but only the public keys.

OpenSSL can also be used to generate RSA keys.

```
openssl genpkey -algorithm rsa -out kf.pem
```

which has the first line:

```
-----BEGIN PRIVATE KEY-----
```

This is PKCS8. Other options include size and text output (including the modulus and so on).

```
-pkeyopt rsa_keygen_bits:2048
-text
```

Using `-text``with ``genpkey`

```
> openssl genpkey -algorithm rsa -text
.............................+++
.................+++
-----BEGIN PRIVATE KEY-----
MIIEvgIBADANBgkqhkiG9w0BAQEFAASCBKgwggSkAgEAAoIBAQDML5loXhr+lCXN
..
-----END PRIVATE KEY-----
Private-Key: (2048 bit)
modulus:
    00:cc:2f:99:68:5e:1a:fe:94:25:cd:b0:85:19:69:
    ..
publicExponent: 65537 (0x10001)
privateExponent: ..
prime1: ..
prime2: ..
exponent1: ..
exponent2: ..
coefficient: ..
```

Can we use `-text` to get these values for a pre-existing key? When I run

```
openssl rsa -text -in kf.pem
```

where there is an existing key `kf.pem`, I get output, but it looks different, Output:

```
> openssl rsa -text -in kf.pem
Private-Key: (2048 bit)
..
-----BEGIN RSA PRIVATE KEY-----
..
```

What's going on is that `openssl rsa` has read the PKCS8 format and reformatted as PKCS1, and that's why the base64 sections don't match.

However, it will not read the OpenSSH formatted private key.

3.8 OpenSSL: read PKCS8

We used

```
> openssl genpkey -algorithm rsa
```

and ended up with a PKCS8 formatted key in `kf.pem`. We want to read this key and extract the data, and compare that with what we get with the `openssl rsa` utility. Let's do that part first.

```
> openssl rsa -text -in kf.pem
```

What we get is text formatted as hex bytes with ':' separators.

```
Private-Key: (2048 bit)
modulus:
    00:f4:26:20:6d:d6:08:fb:03:3a:cb:2c:8e:f1:52:
    52:b2:20:39:d4:85:2d:33:33:52:ad:f2:36:85:92:
    af:77:e2:9e:46:cb:78:aa:79:f9:df:4c:d8:51:bb:
    c3:86:e3:ea:f7:81:98:bc:a8:00:39:0f:92:57:49:
    32:7b:fc:61:d1:0b:ac:25:f1:8b:b3:ed:70:72:f2:
    7e:8a:47:99:28:aa:20:d5:11:88:0e:c8:34:e4:ef:
    e7:d7:4e:04:4d:e0:3b:c6:e9:dc:55:60:27:d1:11:
    ac:06:55:60:c3:07:84:ec:41:8b:09:ed:0c:e4:18:
    36:b7:a8:b1:b2:05:c7:75:69:69:2a:25:3d:3f:ff:
    db:4c:17:94:19:ba:7c:aa:0c:01:9e:bc:a7:5f:53:
    59:15:3a:4c:41:9d:4b:80:1b:6d:5e:23:cd:72:76:
    19:8d:d4:f6:76:f1:a2:c4:a5:07:52:2f:88:68:cd:
    4f:65:ac:37:f2:1f:09:f7:73:71:3f:3c:79:2a:c1:
    91:83:23:4e:b8:02:d2:8c:ed:26:67:9f:92:dd:d9:
    49:04:37:59:63:9f:bc:0b:e0:fd:cd:dc:10:92:fe:
    e2:de:bc:6f:a7:9f:52:7b:74:3f:10:82:63:6c:90:
    cf:a8:c7:06:c1:96:39:f1:c6:fc:5b:fa:87:c2:4f:
    29:15
publicExponent: 65537 (0x10001)
privateExponent:
    00:af:b9:a9:69:a4:bd:fd:fd:0b:1a:25:4e:14:ff:
    4d:aa:0b:6b:d4:3c:ae:95:c5:80:e2:d6:0c:cc:03:
    11:ec:55:dd:d9:d2:a5:5c:fe:42:0c:a8:c0:a1:c3:
    65:2d:f7:69:ad:0f:48:21:b1:41:c7:d0:1f:62:57:
    ba:d0:66:8c:f8:eb:4f:d2:57:92:57:c4:b4:44:e7:
```

(continues on next page)

(continued from previous page)

```
    a8:90:5a:8c:30:2a:93:4b:08:3d:47:76:6e:2b:c1:
    48:bb:3c:d9:f8:3b:46:8b:1a:d3:8d:57:92:10:f6:
    89:3c:5d:c3:31:5c:7e:1d:95:e7:3f:13:b1:4d:92:
    e7:ff:34:9a:01:2a:0b:af:c2:f1:1d:9c:5b:06:ff:
    b2:38:4e:30:34:86:5e:4d:e9:74:cf:31:fa:98:e4:
    9e:60:dd:4a:01:e8:50:ab:d8:d3:77:0c:a9:ba:e3:
    10:40:6d:19:d5:23:82:51:f2:24:04:b2:d7:0c:b7:
    2d:00:a3:e6:20:9d:4d:4e:f5:ed:91:e6:33:06:6d:
    cf:eb:d8:cb:7c:5e:82:d7:ff:1d:61:0f:85:2f:8f:
    1e:8f:55:3c:bf:2b:e8:b5:d4:d4:b6:30:f9:4a:6a:
    e7:85:19:ac:8b:41:d6:fb:75:63:33:00:55:95:e2:
    29:62:53:2b:6b:81:1b:3e:ef:3d:0d:69:12:19:15:
    04:c1
prime1:
    00:fa:b8:d0:8e:d6:34:c0:c9:2e:ea:33:b7:8b:9b:
    31:5c:b8:d9:48:31:4b:ea:52:13:f3:50:18:8a:d1:
    d8:18:a3:2b:52:04:4e:7c:0d:3d:32:d2:0f:c8:eb:
    88:f1:66:f5:bb:60:a2:62:29:86:5b:82:67:44:bd:
    bf:4a:76:bf:fc:58:9c:8c:91:51:54:a1:4f:dc:ab:
    44:df:b8:d5:42:78:8f:6b:c6:31:6e:71:d2:d5:57:
    6e:5a:93:a3:af:7e:ae:28:87:4e:0d:02:e3:3f:c5:
    c0:52:b9:29:34:f3:e3:dc:f9:ae:55:5c:19:07:a0:
    d8:86:aa:13:33:d2:5b:a0:6d
prime2:
    00:f9:49:e3:90:88:e3:1e:1d:70:aa:7d:d9:97:c7:
    88:09:df:5f:0c:26:6c:a4:84:10:ea:8c:52:57:85:
    d8:17:fa:ac:95:91:ed:28:89:dd:af:c4:b8:bc:d3:
    95:14:4e:cf:52:2f:49:1a:74:88:08:d9:81:ec:20:
    41:20:00:07:7e:8b:16:62:bf:f3:90:5c:65:18:78:
    b7:b6:e2:20:63:43:f9:90:24:2e:14:14:90:37:12:
    6a:9d:00:98:9d:e3:86:21:b5:21:60:27:1d:77:ca:
    13:0e:dd:e6:c8:c6:96:ea:e8:dc:3f:15:e6:ec:bd:
    4f:42:4b:c9:00:e3:c9:d2:49
exponent1:
    3a:54:6a:f9:00:2e:cf:b7:3e:79:f0:44:40:6f:7f:
    a1:71:c3:e3:3e:cc:c9:9c:04:d6:33:89:32:2a:b5:
    da:ad:83:73:96:5a:e8:13:70:6c:75:60:84:be:ff:
    62:22:31:03:41:ed:25:67:41:c1:e2:69:c2:1d:5e:
    f6:a4:ff:ef:66:72:2d:65:d5:85:19:ee:69:89:53:
    01:b5:8f:af:e2:3a:83:b9:5d:60:b3:8c:78:63:d9:
    e1:aa:bd:87:23:b2:c2:ed:0f:a4:89:4a:73:58:bf:
    70:bf:71:2d:c7:9b:f8:9a:02:0c:0b:dc:2a:e1:29:
    de:d2:8c:9b:1e:d2:80:55
exponent2:
    70:47:b5:75:8e:12:2d:a8:38:ec:b1:8e:65:ec:7a:
    fb:67:5e:5a:0c:9c:76:64:fd:71:87:0e:37:59:93:
    81:09:68:de:5d:41:a2:36:a6:60:da:8c:12:90:81:
    df:09:b8:1b:5e:2c:e0:fb:87:a1:e4:c5:bd:e2:b1:
    32:86:90:d9:90:2f:de:fe:71:e7:9d:95:f3:35:bc:
    19:65:34:0d:41:ba:90:0f:9b:a9:73:b1:98:fc:74:
    84:8e:96:2e:d7:21:bc:e0:e6:4d:76:90:b1:39:94:
    e7:e7:4e:61:34:01:19:81:14:62:5d:ad:0b:08:21:
    40:cc:fd:95:a7:03:69:f1
coefficient:
    5a:2c:5d:98:80:58:04:ac:a7:6f:b2:a6:e5:c9:26:
    96:86:c3:06:75:18:50:14:b4:bd:25:65:c0:2c:32:
    06:3c:8c:b3:da:94:2b:63:d6:34:15:20:7a:64:87:
```

(continues on next page)

(continued from previous page)

```
    4f:a1:5f:ec:9f:c8:96:17:ca:7e:1f:8f:aa:79:bf:
    df:f0:a8:fb:78:e7:71:62:0c:50:19:fb:26:02:e0:
    d2:cc:51:2f:9f:64:f4:d4:be:f0:a1:f3:53:e9:b1:
    bf:e6:83:2d:10:e8:2e:d1:a2:d6:d8:f5:9f:1d:72:
    a0:4f:c3:75:48:73:07:8e:08:cc:df:ac:15:72:66:
    1b:05:6c:ac:bd:e5:30:cc
writing RSA key
-----BEGIN RSA PRIVATE KEY-----
...
```

I wrote a little Python:

```
import sys
s = sys.argv[1]

def f(s):
    s = s.replace(' ', '')
    s = s.replace(':', '')
    s = s.replace('\n', '')
    print int(s,16)

f(s)
```

which I invoke from the command line, pasting the hex

```
python decode.py '00:... '
```

modulus

```
> python decode.py '00:f4:26:20:6d:d6:08:fb:03:3a:cb:2c:8e:f1:52:
    52:b2:20:39:d4:85:2d:33:33:52:ad:f2:36:85:92:
    af:77:e2:9e:46:cb:78:aa:79:f9:df:4c:d8:51:bb:
    c3:86:e3:ea:f7:81:98:bc:a8:00:39:0f:92:57:49:
    32:7b:fc:61:d1:0b:ac:25:f1:8b:b3:ed:70:72:f2:
    7e:8a:47:99:28:aa:20:d5:11:88:0e:c8:34:e4:ef:
    e7:d7:4e:04:4d:e0:3b:c6:e9:dc:55:60:27:d1:11:
    ac:06:55:60:c3:07:84:ec:41:8b:09:ed:0c:e4:18:
    36:b7:a8:b1:b2:05:c7:75:69:69:2a:25:3d:3f:ff:
    db:4c:17:94:19:ba:7c:aa:0c:01:9e:bc:a7:5f:53:
    59:15:3a:4c:41:9d:4b:80:1b:6d:5e:23:cd:72:76:
    19:8d:d4:f6:76:f1:a2:c4:a5:07:52:2f:88:68:cd:
    4f:65:ac:37:f2:1f:09:f7:73:71:3f:3c:79:2a:c1:
    91:83:23:4e:b8:02:d2:8c:ed:26:67:9f:92:dd:d9:
    49:04:37:59:63:9f:bc:0b:e0:fd:cd:dc:10:92:fe:
    e2:de:bc:6f:a7:9f:52:7b:74:3f:10:82:63:6c:90:
    cf:a8:c7:06:c1:96:39:f1:c6:fc:5b:fa:87:c2:4f:
    29:15'
308209..447061
>
```

Here, and further on, I have deleted the internal part of the decimal output.

prime1

```
> python decode.py '00:fa:b8:d0:8e:d6:34:c0:c9:2e:ea:33:b7:8b:9b:
    31:5c:b8:d9:48:31:4b:ea:52:13:f3:50:18:8a:d1:
    d8:18:a3:2b:52:04:4e:7c:0d:3d:32:d2:0f:c8:eb:
```

(continues on next page)

(continued from previous page)

```
    88:f1:66:f5:bb:60:a2:62:29:86:5b:82:67:44:bd:
    bf:4a:76:bf:fc:58:9c:8c:91:51:54:a1:4f:dc:ab:
    44:df:b8:d5:42:78:8f:6b:c6:31:6e:71:d2:d5:57:
    6e:5a:93:a3:af:7e:ae:28:87:4e:0d:02:e3:3f:c5:
    c0:52:b9:29:34:f3:e3:dc:f9:ae:55:5c:19:07:a0:
    d8:86:aa:13:33:d2:5b:a0:6d'
176062..413933
>
```

prime2

```
> python decode.py '00:f9:49:e3:90:88:e3:1e:1d:70:aa:7d:d9:97:c7:
    88:09:df:5f:0c:26:6c:a4:84:10:ea:8c:52:57:85:
    d8:17:fa:ac:95:91:ed:28:89:dd:af:c4:b8:bc:d3:
    95:14:4e:cf:52:2f:49:1a:74:88:08:d9:81:ec:20:
    41:20:00:07:7e:8b:16:62:bf:f3:90:5c:65:18:78:
    b7:b6:e2:20:63:43:f9:90:24:2e:14:14:90:37:12:
    6a:9d:00:98:9d:e3:86:21:b5:21:60:27:1d:77:ca:
    13:0e:dd:e6:c8:c6:96:ea:e8:dc:3f:15:e6:ec:bd:
    4f:42:4b:c9:00:e3:c9:d2:49'
175056..689417
>
```

Check that the multiplication works. It does!

```
>>> 1760629..413933 * 175056..689417
308209..447061L
```

e is:

```
publicExponent: 65537 (0x10001)
```

private exponent

```
> python decode.py '00:af:b9:a9:69:a4:bd:fd:fd:0b:1a:25:4e:14:ff:
    4d:aa:0b:6b:d4:3c:ae:95:c5:80:e2:d6:0c:cc:03:
    11:ec:55:dd:d9:d2:a5:5c:fe:42:0c:a8:c0:a1:c3:
    65:2d:f7:69:ad:0f:48:21:b1:41:c7:d0:1f:62:57:
    ba:d0:66:8c:f8:eb:4f:d2:57:92:57:c4:b4:44:e7:
    a8:90:5a:8c:30:2a:93:4b:08:3d:47:76:6e:2b:c1:
    48:bb:3c:d9:f8:3b:46:8b:1a:d3:8d:57:92:10:f6:
    89:3c:5d:c3:31:5c:7e:1d:95:e7:3f:13:b1:4d:92:
    e7:ff:34:9a:01:2a:0b:af:c2:f1:1d:9c:5b:06:ff:
    b2:38:4e:30:34:86:5e:4d:e9:74:cf:31:fa:98:e4:
    9e:60:dd:4a:01:e8:50:ab:d8:d3:77:0c:a9:ba:e3:
    10:40:6d:19:d5:23:82:51:f2:24:04:b2:d7:0c:b7:
    2d:00:a3:e6:20:9d:4d:4e:f5:ed:91:e6:33:06:6d:
    cf:eb:d8:cb:7c:5e:82:d7:ff:1d:61:0f:85:2f:8f:
    1e:8f:55:3c:bf:2b:e8:b5:d4:d4:b6:30:f9:4a:6a:
    e7:85:19:ac:8b:41:d6:fb:75:63:33:00:55:95:e2:
    29:62:53:2b:6b:81:1b:3e:ef:3d:0d:69:12:19:15:
    04:c1'
221832..747393
>
```

Now I check that the math works:

```
>>> e = 65537
>>> d = 221832..747393
>>> n = 308209..447061
>>> m = 18096537
>>>
>>> c = pow(m,e,n)
>>> pow(c,d,n)
18096537L
>>>
```

Looks good. I could check the derivation of the private exponent but I won't bother.

In the next part, we want to decode the data in the private key manually.

3.9 Key conversions

3.9.1 OpenSSH export

Start by generating a new OpenSSH private key:

```
ssh-keygen -N "" -f kf.openssh
-----BEGIN OPENSSH PRIVATE KEY-----
```

which generates both private and public keys.

To generate/convert a *public key* from the OpenSSH private key

- in SSH2 format (RFC4716)

```
ssh-keygen -f kf.openssh -e
---- BEGIN SSH2 PUBLIC KEY ----
```

- in PEM (that is, PKCS1) format

```
ssh-keygen -f kf.openssh -e -m pem
-----BEGIN RSA PUBLIC KEY-----
```

- in PKCS8 format

```
ssh-keygen -f kf.openssh -e -m pkcs8
-----BEGIN PUBLIC KEY-----
```

However, the OpenSSH private key is another matter. One suggested solution is `putty`.

https://federicofr.wordpress.com/2019/01/02/how-to-convert-openssh-private-keys-to-rsa-pem/

I got `putty` from Homebrew.

```
puttygen kf.openssh -O private-openssh -o kf.pkcs1
-----BEGIN RSA PRIVATE KEY----
```

```
alias look="head -n 1"
look
```

3.9.2 PKCS1 to/from 8

1 -> 8

```
openssl pkcs8 -topk8 -inform PEM -outform PEM \
-nocrypt -in kf.pkcs1 -out kf.pkcs8
-----BEGIN PRIVATE KEY-----
```

The `-topk8` flag means "read traditional format private key and write PKCS8."

One can verify the same info in the pkcs1 and pkcs8 versions with

```
openssl rsa -text -in kf.pkcs1
openssl rsa -text -in kf.pkcs8
```

8 -> 1

Also, the latter outputs PKCS1 format(even starting from PKCS8) as well as the modulus, etc.

Although the above won't read an OpenSSH key, because of the data length markers, it's easy to find the pieces in the OpenSSH key. So for example, I find

- OpenSSH, first part of the modulus

```
b70ca0491e6c4b48615b89a23c84158bd629546c916f872e
```

- openssl rsa output:

```
Private-Key: (2048 bit)
modulus:
00:b7:0c:a0:49:1e:6c:4b:48:61:5b:89:a2:3c:84:
15:8b:d6:29:54:6c:91:6f:87:2e:67:c7:41:6f:d2:
```

That looks like it all matches up.

3.9.3 OpenSSH import

```
ssh-keygen -i -f kf -m pkcs8
                    pem
```

Chokes on the ones I tried.

3.10 ASN.1, DER, PEM

3.10.1 ASN.1

wikipedia:

"Abstract Syntax Notation One (ASN.1) is a standard interface description language for defining data structures that can be serialized and deserialized in a cross-platform way. It is broadly used in telecommunications and computer networking, and especially in cryptography."

example:

```
FooProtocol DEFINITIONS ::= BEGIN

    FooQuestion ::= SEQUENCE {
        trackingNumber INTEGER,
        question       IA5String
    }

    FooAnswer ::= SEQUENCE {
        questionNumber INTEGER,
        answer         BOOLEAN
    }

END
```

3.10.2 DER

Any particular example must be encoded. So there are a variety of codecs or encoding and decoding rules including the very common: DER, distinguished encoding rules.

example:

```
FooProtocol DEFINITIONS ::= BEGIN

    FooQuestion ::= SEQUENCE {
        trackingNumber INTEGER(0..199),
        question       IA5String
    }

    FooAnswer ::= SEQUENCE {
        questionNumber INTEGER(10..20),
        answer         BOOLEAN
    }

    FooHistory ::= SEQUENCE {
        questions SEQUENCE(SIZE(0..10)) OF FooQuestion,
        answers   SEQUENCE(SIZE(1..10)) OF FooAnswer,
        anArray   SEQUENCE(SIZE(100))  OF INTEGER(0..1000),
        ...
    }

END
```

The DER version of an RSA key will contain binary data. As a result, another format is very common, namely

3.10.3 PEM

The PEM format is often used to encapsulate DER-encoded ASN.1 certificates and keys in an ASCII-only format. The PEM version of a DER message consists of the base64 encoding of the DER message, preceded by "——BEGIN FOO——" and followed by "——END FOO——," where "FOO" may indicate "CERTIFICATE," "PUBLIC KEY," "PRIVATE KEY" or many other types of content.

3.10.4 Parser

OpenSSL has an ASN.1 parser. But it doesn't seem to work on OpenSSH keys. And in any case, it's not real informative:

```
> openssl genpkey -algorithm rsa > kf
.............................................+++
.................+++
> openssl asn1parse -inform pem -in kf
    0:d=0  hl=4 l=1213 cons: SEQUENCE
    4:d=1  hl=2 l=   1 prim: INTEGER           :00
    7:d=1  hl=2 l=  13 cons: SEQUENCE
    9:d=2  hl=2 l=   9 prim: OBJECT            :rsaEncryption
   20:d=2  hl=2 l=   0 prim: NULL
   22:d=1  hl=4 l=1191 prim: OCTET STRING      [HEX DUMP]:308204A3
```

3.11 RSA key structure: OpenSSH

3.11.1 OpenSSH

We're going to look at key structure, starting with an OpenSSH key.

One way to decode the data in a key is to just enter this command in Terminal (with no return yet):

```
echo "" | base64 -d | hexdump
```

and then paste the whole base64 data string between the quotation marks.

In the case of the OpenSSH public key that I have in `kf.pub` (with first line `ssh-rsa`), I obtain:

```
0000000 00 00 00 07 73 73 68 2d 72 73 61 00 00 00 03 01
0000010 00 01 00 00 01 01 00 cc 41 ed ca f6 cc ad a8 c5
..
```

I found the answer to getting continous hex:

```
echo "" | base64 -d | xxd -p
```

```
000000077373682d727361000000030100010000010100b70ca0491e6c4b
48615b89a23c84158bd629546c916f872e67c7416fd2f1cbf050d4a7821c
f277ded2954e89474d27534f0a121103a1bb5b9471acaaebb95c1b3e7446
17a498a8a91b612c89dd2c8ff9528cd33ea1c05ca58cab86e5f9050f1ffc
a8e8bbc4815b4b810c37922237c788a7dfdb55748f2e3511e95567105b07
0a963f112f0328f84d4985b021b5a6d85eeb3cb0a08d1282cfb4796ef394
3d4ebb03b6b95aab38a8da9b097607b7ac128b496a9e8a81b0d8d0d4fe3d
b60b902c50fa74a12a810cc77fbe18124a6aaa3cacd7b829d048183f7e36
ce77117a3a04f43476feb782e4e79ae65a3ab4a5508806d58286ddcbcbdb
6c96dc2da0d02d988f
```

The first part is a 32-bit int which says the length of the following data section is 7 bytes, followed by those 7 bytes, which is ASCII for ssh-rsa (run hexdump with -C to see this):

```
00 00 00 07
            73 73 68 2d 72 73 61
```

then another 32-bit int for the following data section of 3 bytes

```
00 00 00 03
            01 00 01
```

this is 65537 in hex: `1 * 16**4 + 1`.

Then another 32-bit int:

```
00 00 01 01
```

which says the following data section is 257 bytes.

If we strip off the leading null, that's the length that we should have for a 2048 bit key.

```
..................................................b70ca0491e6c4b
48615b89a23c84158bd629546c916f872e67c7416fd2f1cbf050d4a7821c
f277ded2954e89474d27534f0a121103a1bb5b9471acaaebb95c1b3e7446
17a498a8a91b612c89dd2c8ff9528cd33ea1c05ca58cab86e5f9050f1ffc
a8e8bbc4815b4b810c37922237c788a7dfdb55748f2e3511e95567105b07
0a963f112f0328f84d4985b021b5a6d85eeb3cb0a08d1282cfb4796ef394
3d4ebb03b6b95aab38a8da9b097607b7ac128b496a9e8a81b0d8d0d4fe3d
b60b902c50fa74a12a810cc77fbe18124a6aaa3cacd7b829d048183f7e36
ce77117a3a04f43476feb782e4e79ae65a3ab4a5508806d58286ddcbcbdb
6c96dc2da0d02d988f
```

We can use the script `decode.py` from before, just paste in the relevant bytes (in quotes), obtaining (I have broken the lines)

```
231078359784953640556250936100765278034044456796567803307879
741903952616004667698467445611485962430664366862543283853578
114440818626659592152360008282153539987046965902690291423105
853129532115344940489982148622824657938803130717168030363186
452392494555837804339938659537931611124829371134569636259862
608973210794911874003357425470781233434251847210990719978682
756380369293842875814280516186706300902649573228547539514910
305327885065850259588188003492877601723927652894332863523652
475636717690954561480395722140304918026021402089792935224691
290319419331992767287916398917302730494421618111767072007753
15718739157948559
```

We will refer to this number as 231..559. I have also pasted the original output into the Python interpreter and assigned it to the variable *n*.

Now, of course, we want to check this against the private key.

Again,

```
echo "" | base64 -d | xxd -p
```

```
6f70656e7373682d6b65792d763100000000046e6f6e65000000046e6f6e
6500000000000000010000011700000007737368d727361000000030100
010000010100b70ca0491e6c4b48615b89a23c84158bd629546c916f872e
67c7416fd2f1cbf050d4a7821cf277ded2954e89474d27534f0a121103a1
bb5b9471acaaebb95c1b3e744617a498a8a91b612c89dd2c8ff9528cd33e
a1c05ca58cab86e5f9050f1ffca8e8bbc4815b4b810c37922237c788a7df
db55748f2e3511e95567105b070a963f112f0328f84d4985b021b5a6d85e
```

(continues on next page)

(continued from previous page)

```
eb3cb0a08d1282cfb4796ef3943d4ebb03b6b95aab38a8da9b097607b7ac
128b496a9e8a81b0d8d0d4fe3db60b902c50fa74a12a810cc77fbe18124a
6aaa3cacd7b829d048183f7e36ce77117a3a04f43476feb782e4e79ae65a
3ab4a5508806d58286ddcbcbdb6c96dc2da0d02d988f000003d06002a68f
6002a68f0000000077373682d7273610000010100b70ca0491e6c4b48615b
89a23c84158bd629546c916f872e67c7416fd2f1cbf050d4a7821cf277de
d2954e89474d27534f0a121103a1bb5b9471acaaebb95c1b3e744617a498
a8a91b612c89dd2c8ff9528cd33ea1c05ca58cab86e5f9050f1ffca8e8bb
c4815b4b810c37922237c788a7dfdb55748f2e3511e95567105b070a963f
112f0328f84d4985b021b5a6d85eeb3cb0a08d1282cfb4796ef3943d4ebb
03b6b95aab38a8da9b097607b7ac128b496a9e8a81b0d8d0d4fe3db60b90
2c50fa74a12a810cc77fbe18124a6aaa3cacd7b829d048183f7e36ce7711
7a3a04f43476feb782e4e79ae65a3ab4a5508806d58286ddcbcbdb6c96dc
2da0d02d988f00000003010001000001003c9e32ca6407cada7a5b3cf5fc
0265bc3c3ccd97728633871b98f1c39d60b37faed4ed6ffa34159c35b27a
229df91fe7c7c9f6f7a9733abf76263adf1356fbf88db325af18b3f14ea7
21840557b8352984bbdc0ce6b5f43862a03ef9138128641860bb49cb2667
5c56acfc0e29c4bd10353fbdbbcbf0ca858a9bf1cb42b0526110ae4b0315
2c555d06ff2bd318477c45a2caf77672e732a79209602e669f6ebc5492f5
f8a103c72b24b6c203d4211e4b84cded77155968df361b86ced780a66b7c
2fe7943e395ed8b11cc8adad61e784f62b41559bc4bd23cb25af347e98c7
5c3564d553de6984482f47fd875eee244c5edfc0449333a526fe4d6fbf54
2886a900000080402f0f3ac4ad8b61c36d6e6e0893c5476006dc2d54395b
a58bf31f43731733bcca0c148bc2f9129bf49949b74859850be471936179
99f3465a86aea1ccc67fd79c0d075cbafd1d1a0c3ff52eb80763a392e0af
024a3c3c6cb81eb57aa5b26f5e0edb1571e1038d8742e823cdd54473a0e8
291851612162f8da943bc3a12b54300000008100e075ab32affcb508cba9
28cc1e7e53727435793f4033dda2949d6325ebab8617deff0b306127237c
cb9357cee094177f2242dac37d0619ef89ffa9dcfc43bd3b25233e13713b
107ba7fdceb2331499fe032b7771ce48e25af2c70e38875600b0f3f76dcd
c25d91226ccf7f22370801c8d1844536d0a7c193345e4b05807f40fd0000
008100d0c55520133f638f4582f171c7de06036b1e8b8667a964ae3e26bb
b9c2f85196485637f75e2cd6612c6c78f2682e23bc1b57b8f5f17e490ee8
2379a53ff31b52b70bb998e8fe1cdbc4fa864810b60d0ba44a8d88718076
cf7d6eb91d4b7bf7d792a16f3d7af726e402372cc528849ecb6e5a4bbed6
865db0a0246d3e41c08b7b0000001674656c6c696f7474406e65776d696e
692e6c6f63616c0102030405
```

First, a header that includes 73-73-68-2d (ssh-) but not 72-73-61(rsa):

```
6f70656e7373682d6b65792d7631
```

size and data block of unknown function (conserved across different keys)

```
............................00000000046e6f6e65
```

repeated

```
..............................................000000046e6f6e
65
```

Something I haven’t yet identified:

```
..000000000000000100000117
```

ssh-rsa (this is the public key):

```
..........................00000007
..................................7373682d727361
```

e = 65537:

```
................................................00000003
........................................................0100
01
```

The size and data for *n*, the modulus:

```
..00000101
..........00b70ca0491e6c4b48615b89a23c84158bd629546c916f872e
67c7416fd2f1cbf050d4a7821cf277ded2954e89474d27534f0a121103a1
bb5b9471acaaebb95c1b3e744617a498a8a91b612c89dd2c8ff9528cd33e
a1c05ca58cab86e5f9050f1ffca8e8bbc4815b4b810c37922237c788a7df
db55748f2e3511e95567105b070a963f112f0328f84d4985b021b5a6d85e
eb3cb0a08d1282cfb4796ef3943d4ebb03b6b95aab38a8da9b097607b7ac
128b496a9e8a81b0d8d0d4fe3db60b902c50fa74a12a810cc77fbe18124a
6aaa3cacd7b829d048183f7e36ce77117a3a04f43476feb782e4e79ae65a
3ab4a5508806d58286ddcbcbdb6c96dc2da0d02d988f
```

Stuff of unknown function

```
............................................000003d06002a68f
6002a68f
```

Size and data block that says ssh-rsa

```
........00000007
................7373682d727361
```

The modulus data, repeated

```
................................00000101
........................................b70ca0491e6c4b48615b
89a23c84158bd629546c916f872e67c7416fd2f1cbf050d4a7821cf277de
d2954e89474d27534f0a121103a1bb5b9471acaaebb95c1b3e744617a498
a8a91b612c89dd2c8ff9528cd33ea1c05ca58cab86e5f9050f1ffca8e8bb
c4815b4b810c37922237c788a7dfdb55748f2e3511e95567105b070a963f
112f0328f84d4985b021b5a6d85eeb3cb0a08d1282cfb4796ef3943d4ebb
03b6b95aab38a8da9b097607b7ac128b496a9e8a81b0d8d0d4fe3db60b90
2c50fa74a12a810cc77fbe18124a6aaa3cacd7b829d048183f7e36ce7711
7a3a04f43476feb782e4e79ae65a3ab4a5508806d58286ddcbcbdb6c96dc
2da0d02d988f
```

e, repeated:

```
............00000003
....................010001
```

And now, the new stuff. First, a candidate for the private exponent:

```
..........................00000100
..................................3c9e32ca6407cada7a5b3cf5fc
```

(continues on next page)

(continued from previous page)

```
0265bc3c3ccd97728633871b98f1c39d60b37faed4ed6ffa34159c35b27a
229df91fe7c7c9f6f7a9733abf76263adf1356fbf88db325af18b3f14ea7
21840557b8352984bbdc0ce6b5f43862a03ef9138128641860bb49cb2667
5c56acfc0e29c4bd10353fbdbbcbf0ca858a9bf1cb42b0526110ae4b0315
2c555d06ff2bd318477c45a2caf77672e732a79209602e669f6ebc5492f5
f8a103c72b24b6c203d4211e4b84cded77155968df361b86ced780a66b7c
2fe7943e395ed8b11cc8adad61e784f62b41559bc4bd23cb25af347e98c7
5c3564d553de6984482f47fd875eee244c5edfc0449333a526fe4d6fbf54
2886a9
```

something that turns out not to be either of our primes:

```
......00000080
..............402f0f3ac4ad8b61c36d6e6e0893c5476006dc2d54395b
a58bf31f43731733bcca0c148bc2f9129bf49949b74859850be471936179
99f3465a86aea1ccc67fd79c0d075cbafd1d1a0c3ff52eb80763a392e0af
024a3c3c6cb81eb57aa5b26f5e0edb1571e1038d8742e823cdd54473a0e8
291851612162f8da943bc3a12b5430
```

prime1 aka *p*:

```
..............................00000081
........................................00e075ab32affcb508cba9
28cc1e7e53727435793f4033dda2949d6325ebab8617deff0b306127237c
cb9357cee094177f2242dac37d0619ef89ffa9dcfc43bd3b25233e13713b
107ba7fdceb2331499fe032b7771ce48e25af2c70e38875600b0f3f76dcd
c25d91226ccf7f22370801c8d1844536d0a7c193345e4b05807f40fd
```

prime2 aka *q*:

```
                                                        0000
0081
....00d0c55520133f638f4582f171c7de06036b1e8b8667a964ae3e26bb
b9c2f85196485637f75e2cd6612c6c78f2682e23bc1b57b8f5f17e490ee8
2379a53ff31b52b70bb998e8fe1cdbc4fa864810b60d0ba44a8d88718076
cf7d6eb91d4b7bf7d792a16f3d7af726e402372cc528849ecb6e5a4bbed6
865db0a0246d3e41c08b7b
```

followed by 22 bytes of data

```
......................0000001674656c6c696f7474406e65776d696e
692e6c6f63616c01
```

and finally:

```
................01020304
```

In the first part, we obtained this for the modulus *n*:

```
231..559
```

So to do the check I copied values from above for *p* and *q* and then ran `python decode.py` on each one and then pasted into the interpreter to check the multiplication.

The 81-byte value labeled “something?” was a candidate for prime1, but it turns out the next two values are prime1 and prime2

```
00e075.. = 157..677
00d0c5.. = 146..667
```

The product of these two *is* equal to the modulus.

```
>>> p*q==n
True
```

So if `3c9e..` is *d* equal to `765..577`, then we can paste the full decimal output in and check `(d*e)%phi(n) = 1`

```
>>> phi = (p-1)*(q-1)
>>> e = 65537
>>> d*e % phi
1L
```

Yay!

There are some extra bits and pieces including "something?", but this has all checked out pretty nicely.

OpenSSL has a parsing tool but it throws an error with this key:

```
> openssl asn1parse -inform pem -in kf
0:d=0  hl=2 l= 112 cons: appl [ 15 ]
2:d=1  hl=2 l= 110 cons: appl [ 5 ]
Error in encoding..
```

I compared two different new OpenSSH private keys.

The base64 was the same, which after decoding with the `echo "" | base64 -d | xxd -p` command gives:

```
6f70656e7373682d6b65792d763100000000046e6f6e65000000046e6f6e
650000000000000001000001170000000773736 82d727361000000030100
010000010100aecbc2f5a910ad6a95aeb169ca853d58127c
```

Notice that with the `0000010100ae` on the third line, we are into the modulus. So the header data is constant, and then the first digits of the moduli are also the same.

3.12 RSA key structure: PKCS1

3.12.1 PKCS1

We're going to dissect a PKCS1 key. Luckily, I discovered how to convert an OpenSSH *private* key to PKCS1:

```
puttygen kf.openssh -O private-openssh -o kf.pkcs1
```

So we'll paste the base64 data into the following:

```
echo "" | base64 -d | xxd -p
```

```
308204a20201000282010100b70ca0491e6c4b48615b89a23c84158bd629
546c916f872e67c7416fd2f1cbf050d4a7821cf277ded2954e89474d2753
4f0a121103a1bb5b9471acaaebb95c1b3e744617a498a8a91b612c89dd2c
```

(continues on next page)

(continued from previous page)

```
8ff9528cd33ea1c05ca58cab86e5f9050f1ffca8e8bbc4815b4b810c3792
2237c788a7dfdb55748f2e3511e95567105b070a963f112f0328f84d4985
b021b5a6d85eeb3cb0a08d1282cfb4796ef3943d4ebb03b6b95aab38a8da
9b097607b7ac128b496a9e8a81b0d8d0d4fe3db60b902c50fa74a12a810c
c77fbe18124a6aaa3cacd7b829d048183f7e36ce77117a3a04f43476feb7
82e4e79ae65a3ab4a5508806d58286ddcbcbdb6c96dc2da0d02d988f0203
010001028201003c9e32ca6407cada7a5b3cf5fc0265bc3c3ccd97728633
871b98f1c39d60b37faed4ed6ffa34159c35b27a229df91fe7c7c9f6f7a9
733abf76263adf1356fbf88db325af18b3f14ea721840557b8352984bbdc
0ce6b5f43862a03ef9138128641860bb49cb26675c56acfc0e29c4bd1035
3fbdbbcbf0ca858a9bf1cb42b0526110ae4b03152c555d06ff2bd318477c
45a2caf77672e732a79209602e669f6ebc5492f5f8a103c72b24b6c203d4
211e4b84cded77155968df361b86ced780a66b7c2fe7943e395ed8b11cc8
adad61e784f62b41559bc4bd23cb25af347e98c75c3564d553de6984482f
47fd875eee244c5edfc0449333a526fe4d6fbf542886a902818100e075ab
32affcb508cba928cc1e7e53727435793f4033dda2949d6325ebab8617de
ff0b306127237ccb9357cee094177f2242dac37d0619ef89ffa9dcfc43bd
3b25233e13713b107ba7fdceb2331499fe032b7771ce48e25af2c70e3887
5600b0f3f76dcdc25d91226ccf7f22370801c8d1844536d0a7c193345e4b
05807f40fd02818100d0c55520133f638f4582f171c7de06036b1e8b8667
a964ae3e26bbb9c2f85196485637f75e2cd6612c6c78f2682e23bc1b57b8
f5f17e490ee82379a53ff31b52b70bb998e8fe1cdbc4fa864810b60d0ba4
4a8d88718076cf7d6eb91d4b7bf7d792a16f3d7af726e402372cc528849e
cb6e5a4bbed6865db0a0246d3e41c08b7b02818010f0907508d7178a8c64
3dd1f9d32ad50e7422ac655a04b60a653e605442e3a1d59085f5a6bf5f6c
41a8e30d97e1554ca0f74eaf463639aadc4d51327e4f566eaa44d8a07b01
2c2348f05cb3122bd1db5bbde20bd2b7a96d97b8f217ce0eed8a6d3f1528
5c2cd79133890d5d35ae030476db76a5c66582b46b555a7015dc84b90281
800b826580002fa57077979172015fc71b1723b6f370f190c05e62ca44a2
7008276dd37b00632bbba1ddce1918dc0f771edeaf065e60b2e29a34c807
e3c953c0b4ddac82cde0426a1adf90242902ead66b46e2694d155cccb001
ad41fd50750423d94c97125f9e1281cd717741634b7469a8aac7b43ca569
b2dc114608819d78d3028180402f0f3ac4ad8b61c36d6e6e0893c5476006
dc2d54395ba58bf31f43731733bcca0c148bc2f9129bf49949b74859850b
e47193617999f3465a86aea1ccc67fd79c0d075cbafd1d1a0c3ff52eb807
63a392e0af024a3c3c6cb81eb57aa5b26f5e0edb1571e1038d8742e823cd
d54473a0e8291851612162f8da943bc3a12b5430
```

Now we can search for the byte strings we found in the previous chapter.

I easily find *n*, *e*, *d*, *p* and *q*

```
308204a2020100
              02820101
----------------------00b70ca0491e6c4b48615b89a23c84158bd629  n
546c916f872e67c7416fd2f1cbf050d4a7821cf277ded2954e89474d2753
4f0a121103a1bb5b9471acaaebb95c1b3e744617a498a8a91b612c89dd2c
8ff9528cd33ea1c05ca58cab86e5f9050f1ffca8e8bbc4815b4b810c3792
2237c788a7dfdb55748f2e3511e95567105b070a963f112f0328f84d4985
b021b5a6d85eeb3cb0a08d1282cfb4796ef3943d4ebb03b6b95aab38a8da
9b097607b7ac128b496a9e8a81b0d8d0d4fe3db60b902c50fa74a12a810c
c77fbe18124a6aaa3cacd7b829d048183f7e36ce77117a3a04f43476feb7
82e4e79ae65a3ab4a5508806d58286ddcbcbdb6c96dc2da0d02d988f
........................................................0203
010001                                                        e
....................................................02820100. d
..............3c9e32ca6407cada7a5b3cf5fc0265bc3c3ccd97728633
```

(continues on next page)

(continued from previous page)

```
871b98f1c39d60b37faed4ed6ffa34159c35b27a229df91fe7c7c9f6f7a9
733abf76263adf1356fbf88db325af18b3f14ea721840557b8352984bbdc
0ce6b5f43862a03ef9138128641860bb49cb26675c56acfc0e29c4bd1035
3fbdbbcbf0ca858a9bf1cb42b0526110ae4b03152c555d06ff2bd318477c
45a2caf77672e732a79209602e669f6ebc5492f5f8a103c72b24b6c203d4
211e4b84cded77155968df361b86ced780a66b7c2fe7943e395ed8b11cc8
adad61e784f62b41559bc4bd23cb25af347e98c75c3564d553de6984482f
47fd875eee244c5edfc0449333a526fe4d6fbf542886a9
..............................................028181
....................................................00e075ab  p
32affcb508cba928cc1e7e53727435793f4033dda2949d6325ebab8617de
ff0b306127237ccb9357cee094177f2242dac37d0619ef89ffa9dcfc43bd
3b25233e13713b107ba7fdceb2331499fe032b7771ce48e25af2c70e3887
5600b0f3f76dcdc25d91226ccf7f22370801c8d1844536d0a7c193345e4b
05807f40fd
..........028181
................00d0c55520133f638f4582f171c7de06036b1e8b8667  q
a964ae3e26bbb9c2f85196485637f75e2cd6612c6c78f2682e23bc1b57b8
f5f17e490ee82379a53ff31b52b70bb998e8fe1cdbc4fa864810b60d0ba4
4a8d88718076cf7d6eb91d4b7bf7d792a16f3d7af726e402372cc528849e
cb6e5a4bbed6865db0a0246d3e41c08b7b
..................................028180
........................................10f0907508d7178a8c64
3dd1f9d32ad50e7422ac655a04b60a653e605442e3a1d59085f5a6bf5f6c
41a8e30d97e1554ca0f74eaf463639aadc4d51327e4f566eaa44d8a07b01
2c2348f05cb3122bd1db5bbde20bd2b7a96d97b8f217ce0eed8a6d3f1528
5c2cd79133890d5d35ae030476db76a5c66582b46b555a7015dc84b9
........................................................0281
80
..0b826580002fa57077979172015fc71b1723b6f370f190c05e62ca44a2
7008276dd37b00632bbba1ddce1918dc0f771edeaf065e60b2e29a34c807
e3c953c0b4ddac82cde0426a1adf90242902ead66b46e2694d155cccb001
ad41fd50750423d94c97125f9e1281cd717741634b7469a8aac7b43ca569
b2dc114608819d78d3
..................028180
........................402f0f3ac4ad8b61c36d6e6e0893c5476006 ?
dc2d54395ba58bf31f43731733bcca0c148bc2f9129bf49949b74859850b
e47193617999f3465a86aea1ccc67fd79c0d075cbafd1d1a0c3ff52eb807
63a392e0af024a3c3c6cb81eb57aa5b26f5e0edb1571e1038d8742e823cd
d54473a0e8291851612162f8da943bc3a12b5430
```

Each is preceded by a value like

- 02820 + 101
- 02 + 03
- 0281 + 81
- 0281 + 81
- 0281 + 80
- 0281 + 80

There is a header, but no footer.

Using this clue, I divide up the unknown region. Having done that, we can compare with what the rsa tool gives us. (I couldn't load PKCS1, but did get PKCS8 to work).

```
modulus:
    00:b7:0c:a0:49:1e:6c:4b:48:61:5b:89:a2:3c:84:
    15:8b:d6:29:54:6c:91:6f:87:2e:67:c7:41:6f:d2:
    f1:cb:f0:50:d4:a7:82:1c:f2:77:de:d2:95:4e:89:
    47:4d:27:53:4f:0a:12:11:03:a1:bb:5b:94:71:ac:
    aa:eb:b9:5c:1b:3e:74:46:17:a4:98:a8:a9:1b:61:
    2c:89:dd:2c:8f:f9:52:8c:d3:3e:a1:c0:5c:a5:8c:
    ab:86:e5:f9:05:0f:1f:fc:a8:e8:bb:c4:81:5b:4b:
    81:0c:37:92:22:37:c7:88:a7:df:db:55:74:8f:2e:
    35:11:e9:55:67:10:5b:07:0a:96:3f:11:2f:03:28:
    f8:4d:49:85:b0:21:b5:a6:d8:5e:eb:3c:b0:a0:8d:
    12:82:cf:b4:79:6e:f3:94:3d:4e:bb:03:b6:b9:5a:
    ab:38:a8:da:9b:09:76:07:b7:ac:12:8b:49:6a:9e:
    8a:81:b0:d8:d0:d4:fe:3d:b6:0b:90:2c:50:fa:74:
    a1:2a:81:0c:c7:7f:be:18:12:4a:6a:aa:3c:ac:d7:
    b8:29:d0:48:18:3f:7e:36:ce:77:11:7a:3a:04:f4:
    34:76:fe:b7:82:e4:e7:9a:e6:5a:3a:b4:a5:50:88:
    06:d5:82:86:dd:cb:cb:db:6c:96:dc:2d:a0:d0:2d:
    98:8f
publicExponent: 65537 (0x10001)
privateExponent:
    3c:9e:32:ca:64:07:ca:da:7a:5b:3c:f5:fc:02:65:
    bc:3c:3c:cd:97:72:86:33:87:1b:98:f1:c3:9d:60:
    b3:7f:ae:d4:ed:6f:fa:34:15:9c:35:b2:7a:22:9d:
    f9:1f:e7:c7:c9:f6:f7:a9:73:3a:bf:76:26:3a:df:
    13:56:fb:f8:8d:b3:25:af:18:b3:f1:4e:a7:21:84:
    05:57:b8:35:29:84:bb:dc:0c:e6:b5:f4:38:62:a0:
    3e:f9:13:81:28:64:18:60:bb:49:cb:26:67:5c:56:
    ac:fc:0e:29:c4:bd:10:35:3f:bd:bb:cb:f0:ca:85:
    8a:9b:f1:cb:42:b0:52:61:10:ae:4b:03:15:2c:55:
    5d:06:ff:2b:d3:18:47:7c:45:a2:ca:f7:76:72:e7:
    32:a7:92:09:60:2e:66:9f:6e:bc:54:92:f5:f8:a1:
    03:c7:2b:24:b6:c2:03:d4:21:1e:4b:84:cd:ed:77:
    15:59:68:df:36:1b:86:ce:d7:80:a6:6b:7c:2f:e7:
    94:3e:39:5e:d8:b1:1c:c8:ad:ad:61:e7:84:f6:2b:
    41:55:9b:c4:bd:23:cb:25:af:34:7e:98:c7:5c:35:
    64:d5:53:de:69:84:48:2f:47:fd:87:5e:ee:24:4c:
    5e:df:c0:44:93:33:a5:26:fe:4d:6f:bf:54:28:86:
    a9
prime1:
    00:e0:75:ab:32:af:fc:b5:08:cb:a9:28:cc:1e:7e:
    53:72:74:35:79:3f:40:33:dd:a2:94:9d:63:25:eb:
    ab:86:17:de:ff:0b:30:61:27:23:7c:cb:93:57:ce:
    e0:94:17:7f:22:42:da:c3:7d:06:19:ef:89:ff:a9:
    dc:fc:43:bd:3b:25:23:3e:13:71:3b:10:7b:a7:fd:
    ce:b2:33:14:99:fe:03:2b:77:71:ce:48:e2:5a:f2:
    c7:0e:38:87:56:00:b0:f3:f7:6d:cd:c2:5d:91:22:
    6c:cf:7f:22:37:08:01:c8:d1:84:45:36:d0:a7:c1:
    93:34:5e:4b:05:80:7f:40:fd
prime2:
    00:d0:c5:55:20:13:3f:63:8f:45:82:f1:71:c7:de:
    06:03:6b:1e:8b:86:67:a9:64:ae:3e:26:bb:b9:c2:
    f8:51:96:48:56:37:f7:5e:2c:d6:61:2c:6c:78:f2:
    68:2e:23:bc:1b:57:b8:f5:f1:7e:49:0e:e8:23:79:
    a5:3f:f3:1b:52:b7:0b:b9:98:e8:fe:1c:db:c4:fa:
    86:48:10:b6:0d:0b:a4:4a:8d:88:71:80:76:cf:7d:
    6e:b9:1d:4b:7b:f7:d7:92:a1:6f:3d:7a:f7:26:e4:
```

(continues on next page)

(continued from previous page)

```
    02:37:2c:c5:28:84:9e:cb:6e:5a:4b:be:d6:86:5d:
    b0:a0:24:6d:3e:41:c0:8b:7b
exponent1:
    10:f0:90:75:08:d7:17:8a:8c:64:3d:d1:f9:d3:2a:
    d5:0e:74:22:ac:65:5a:04:b6:0a:65:3e:60:54:42:
    e3:a1:d5:90:85:f5:a6:bf:5f:6c:41:a8:e3:0d:97:
    e1:55:4c:a0:f7:4e:af:46:36:39:aa:dc:4d:51:32:
    7e:4f:56:6e:aa:44:d8:a0:7b:01:2c:23:48:f0:5c:
    b3:12:2b:d1:db:5b:bd:e2:0b:d2:b7:a9:6d:97:b8:
    f2:17:ce:0e:ed:8a:6d:3f:15:28:5c:2c:d7:91:33:
    89:0d:5d:35:ae:03:04:76:db:76:a5:c6:65:82:b4:
    6b:55:5a:70:15:dc:84:b9
exponent2:
    0b:82:65:80:00:2f:a5:70:77:97:91:72:01:5f:c7:
    1b:17:23:b6:f3:70:f1:90:c0:5e:62:ca:44:a2:70:
    08:27:6d:d3:7b:00:63:2b:bb:a1:dd:ce:19:18:dc:
    0f:77:1e:de:af:06:5e:60:b2:e2:9a:34:c8:07:e3:
    c9:53:c0:b4:dd:ac:82:cd:e0:42:6a:1a:df:90:24:
    29:02:ea:d6:6b:46:e2:69:4d:15:5c:cc:b0:01:ad:
    41:fd:50:75:04:23:d9:4c:97:12:5f:9e:12:81:cd:
    71:77:41:63:4b:74:69:a8:aa:c7:b4:3c:a5:69:b2:
    dc:11:46:08:81:9d:78:d3
coefficient:
    40:2f:0f:3a:c4:ad:8b:61:c3:6d:6e:6e:08:93:c5:
    47:60:06:dc:2d:54:39:5b:a5:8b:f3:1f:43:73:17:
    33:bc:ca:0c:14:8b:c2:f9:12:9b:f4:99:49:b7:48:
    59:85:0b:e4:71:93:61:79:99:f3:46:5a:86:ae:a1:
    cc:c6:7f:d7:9c:0d:07:5c:ba:fd:1d:1a:0c:3f:f5:
    2e:b8:07:63:a3:92:e0:af:02:4a:3c:3c:6c:b8:1e:
    b5:7a:a5:b2:6f:5e:0e:db:15:71:e1:03:8d:87:42:
    e8:23:cd:d5:44:73:a0:e8:29:18:51:61:21:62:f8:
    da:94:3b:c3:a1:2b:54:30
```

So these other values are called the exponents (1 & 2) and the coefficient.

Here is a reference:

https://www.di-mgt.com.au/crt_rsa.html

`exponent1` is the inverse of `e` mod `(p-1)` and `exponent2` is the inverse of `e` mod `(q-1)` while the `coefficient` is the inverse of `q` mod `p`.

I pasted in the values and checked that the results of calculation match the values given in the hex output above.

We'll explore what these do in another chapter. Basically the answer is that computing the mod with `phi` can be done faster if you know `p` and `q` and something called the Chinese Remainder Theorem.

3.13 Encrypting and decrypting

3.13.1 Asymmetric coding and decoding:

```
echo "hello world" > p.txt
openssl genpkey -algorithm rsa > kf.private.pem
openssl rsa -in kf.private.pem -pubout -out kf.public.pem
openssl rsautl -encrypt -in p.txt -out c.txt -pubin -inkey kf.public.pem
openssl rsautl -decrypt -in c.txt -inkey kf.private.pem
```

output:

```
hello world
```

Notice: we must generate the public key in a second step, and then provide the correct key files for encryption and decryption.

or

```
openssl rsautl -sign -in p.txt -out c.txt -inkey kf.private.pem
openssl rsautl -verify -in c.txt -pubin -inkey kf.public.pem
```

with the same output

Symmetric encryption

Using random data

```
echo "hello world" > p.txt
openssl rand 250 > key
openssl enc -aes-256-cbc -salt -in p.txt -out c.txt -pass file:./key
openssl enc -d -aes-256-cbc -in c.txt -out m.txt -pass file:./key
cat m.txt
```

output:

```
hello world
```

3.14 Python RSA module

https://stuvel.eu/python-rsa-doc

3.14.1 Example

```
> openssl genrsa -out kf.pem 512
Generating RSA private key, 512 bit long modulus
..........++++++++++++
.........++++++++++++
e is 65537 (0x10001)
> pyrsa-priv2pub -i kf.pem -o kf.pub.pem
Reading private key from kf.pem in PEM format
Writing public key to kf.pub.pem in PEM format
> echo "hello world" > x.txt
```

:: > pyrsa-encrypt -i x.txt -o c.bin kf.pub.pem Reading public key from kf.pub.pem Reading input from x.txt Encrypting Writing output to c.bin > openssl rsautl -in c.bin -inkey kf.pem -decrypt hello world >

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search