Applications of RSA

Identification of an individual by challenging the RSA key

Satoshi and Craig (say!) are two different persons, each with their own RSA key pair
Everyone knows Satoshi's public key (by calling getPublic(), no problem)
Craig pretends to be Satoshi

We pick a secret, encrypt it with Satoshi's public key
Let both decrypt it and see who's doing it right

Ignore the signString method for the moment - we'll deal with it in the second example

In [2]:
```
from RSA import Key, KeyPair
from Modular import fastExpMod
import random

class Person :
    def __init__(self):
        self._rsaKeys = KeyPair()

    def getPublic(self):
        return self._rsaKeys.public

    def decrypt(self, message):
        return self._rsaKeys.decrypt(message)

    def signString(self,string) :
        return string, self.decrypt(len(string)) # same as encrypt with the private ke


# Satoshi is the "true" one and we know the public key
satoshi = Person()
satoshiKey = satoshi.getPublic()
print("I know the public key of Satoshi: ",satoshiKey)

# Craig says he's Satoshi - so, we don't need his public key,
# he would give us Satoshi's public key anyway (everyone knows it)
craig = Person()

# We can find out who the real Satoshi is by
# 1. picking a secret message "secret"
# 2. encrypting it with Satoshi's public key
# 3. and asking both to decrypt it
#
# Only the "true" Satoshi will do that correctly!
#
secret = 1234

# Use P1's public key to encrypt the secret
challenge = fastExpMod(secret,satoshiKey.second,satoshiKey.first)
print("Challenging with",challenge)

# Let both decrypt it ==> We know who is Satoshi (the one answering with the secret)
```

```
print(satoshi.decrypt(challenge))
print(craig.decrypt(challenge))
```

```
I know the public key of Satoshi:  Key [first: 162179962939, second: 65537]
Challenging with 120824789655
1234
133735024079
```

Sketch of digital signatures

That relies on the fact that you can *use your private key to encrypt* and *your public key to decrypt* (opposite of normal, but we know that works)

So P1 computes a property of the message (like a hash code, here it's just the length), and encrypts it with the *private* (!) key - that's the whole trick!

The receiver can now use P1's public key to check this property and validate that P1 really signed it

In [3]:
```
# Two persons with their RSA keys, public parts are known
p1 = Person()
p1Key = p1.getPublic()
p2 = Person()
p2Key = p2.getPublic()

# Now P1 signs this string with length 13 encrypted with the private (!) key
# So, the encrypted length of the string is the digital signature!
string, signature = p1.signString("Hello, world!")
print("String =",string,"Signature =",signature)

# Now decrypt the signature with each of the public keys
length1 = fastExpMod(signature,p1Key.second,p1Key.first)
length2 = fastExpMod(signature,p2Key.second,p2Key.first)
# And the authenticity is clear
print("Decrypted with P1's public key:",length1,
      " \t\tAuthentic: ",len(string) == length1)
print("Decrypted with P2's public key:",length2,
      " \tAuthentic: ",len(string) == length2)
```

```
String = Hello, world! Signature = 178296806428
Decrypted with P1's public key: 13              Authentic:  True
Decrypted with P2's public key: 18131702359     Authentic:  False
```

Now we need some moment to clarify something about this text property

I used the length of the string as the property - for simplification purposes!

Now, that means that an evil person could abuse this scheme:

1. Intercept the string with the signature
2. Replace the string with another one of the same length

This string would now appear as being signed by P1, but it's not the authentic string anymore: A string "Attack at dawn" could now be replaced with "Attack at noon" (note that "Attack at night" wouldn't work!) with possibly devastating consequences.

In fact, this scheme allows for signing, but doesn't preserve authenticity!

In practice these things go hand in hand, so we need to find a string property that's not easy to fake.

Hashing (More in "Algorithms and Data Structures")

Hashing means to assign to a data object a numerical value. Different data objects might or might not have different values. The data object itself cannot be reconstructed from this value.

Actually, the length is a kind of such a hash value, it's just not good enough to protect against faking. Strong hash values make it practically impossible to find a string that has the same hash value as the authentic one. It's practically impossible to learn anything about the string given a strong hash:

In [4]:
```
import hashlib

hash_object = hashlib.sha256(b'John')
hex_dig = hash_object.hexdigest()
print(hex_dig)

hash_object = hashlib.sha256(b'Joan')
hex_dig = hash_object.hexdigest()
print(hex_dig)

hash_object = hashlib.sha256(b'There now is your insular city of the Manhattoes, belte
hex_dig = hash_object.hexdigest()
print(hex_dig)
```

```
a8cfcd74832004951b4408cdb0a5dbcd8c7e52d43f7fe244bf720582e05241da
2d0f4c4eb78ce93adc09b60c696c76d0476185983c956a6f2a5bbf0afb9dbc2e
10b6271b99876bc2b3b51d76118e5a5462043f6e92a9d6d445b01735f79042e6
```

You can use hash values to encode passwords:

1. Compute the hash for the password during sign-up
2. For each login attempt compute the hash of the provided password and compare it to the sign-up version

Note that this is **not** foolproof, since there are definitely many strings that have the same hash code. An attacker would not have to find the correct password but any string with the same hash - that's all. These strings are called *collisions* and an attack trying to find a suiting string is called a *collision attack*.

In [8]:
```
import hashlib

password = 's3cr3t' # terrible password
hash_object = hashlib.sha256(password.encode('utf8'))
hex_dig_original = hash_object.hexdigest()
print(hex_dig_original)
attempt = 's3cr3t' # another terrible password
hash_object = hashlib.sha256(attempt.encode('utf8'))
hex_dig_attempt = hash_object.hexdigest()
print(hex_dig_attempt)
```

```
if hex_dig_original == hex_dig_attempt :
    print("accepted!")
else :
    print("rejected!")
```

```
4e738ca5563c06cfd0018299933d58db1dd8bf97f6973dc99bf6cdc64b5550bd
4e738ca5563c06cfd0018299933d58db1dd8bf97f6973dc99bf6cdc64b5550bd
accepted!
```

We "improve" the previous example with a weak hash that just adds the decimal value of the characters together. It's still weak, but it makes it much more difficult to sneak an unauthenticated replacement in - it needs to have the same "hash", not be complete gibberish and express something malicious.

The reason I'm not using SHA here is that this toy implementaion of RSA cannot handle numbers that big. You certainly can work around that, by encrypting the string character character and somehow magically appending this into a value.

In [11]:
```
class SecurePerson (Person):
    @classmethod
    def hashStringBadly(cls,string) :
        sum = 0
        for c in string :
            sum += ord(c)
        return sum

    def signString(self,string) :
        return string, self.decrypt(SecurePerson.hashStringBadly(string)) # same as er

# Same with safe persons:
p1 = SecurePerson()
p1Key = p1.getPublic()
p2 = SecurePerson()
p2Key = p2.getPublic()

string, signature = p1.signString("Hello, world!")
print("String =",string,"Signature =",signature)

# Now decrypt the signature with each of the public keys
hash1 = fastExpMod(signature,p1Key.second,p1Key.first)
hash2 = fastExpMod(signature,p2Key.second,p2Key.first)
# And the authenticity is clear
print("Decrypted with P1's public key:",hash1,
      " \t\tAuthentic: ",SecurePerson.hashStringBadly(string) == hash1)
print("Decrypted with P2's public key:",hash2,
      " \tAuthentic: ",SecurePerson.hashStringBadly(string) == hash2)
```

```
String = Hello, world! Signature = 27991652002
Decrypted with P1's public key: 1161            Authentic:  True
Decrypted with P2's public key: 9593639908      Authentic:  False
```

A look at the standard RSA package in Python

Nothing too special, one little thing that's not textbook: Public and private key are NOT interchangeable! The reason is (I assume) that the decryption method checks if the cyphertext

has been properly encrypted with the public key (which is a part of the private key, if I see that correctly).

That means you cannot really use it for Digital Signatures the way it was sketched, requiring a separate functionality to do that.

In [13]:
```
import rsa

# Basic RSA encryption/decryption

# Key pair generation
(pubkey, privkey) = rsa.newkeys(2048)
print(pubkey,"\n",privkey,"\n\n")

# Encryption/Decryption
message = 'Hello, World!'.encode('utf8')
crypto = rsa.encrypt(message, pubkey)
print("Cyphertext:",crypto)
decrypt = rsa.decrypt(crypto, privkey)
print(message.decode('utf8'),"\n\n")

# Digital signature is extra, because (apparently) public and private key are not inte
# Important is that the message is signed with the private key (as it should)
signature = rsa.sign(message, privkey, 'SHA-256')

# Verify with the public key!
print("Signature:",signature,"\nVerified as",rsa.verify(message, signature, pubkey))

# Wrong signature: Have to catch the exception (OHHH-KAYYY)
signature = "12345".encode('utf-8')
try:
    print("Signature:",signature,"\nVerified as",rsa.verify(message, signature, pubkey
except rsa.VerificationError:
    print("Invalid signature")
```

```
PublicKey(19635147346358152523585460051520399747938909830901638291391982237612334909815950385378223723722675271661075436948247200427849953793496917425333564529654748960658671796071297665921133521300063356695163845023427299490230537465582736678491096431339216699040119170682280165568433718690904532890048892899702012215187115758875913521261498134287362191061174921921779724376492690190041440244826539380251009567864619183131345045332312788159686416082694806399559912848040459746017339253978818870208587143877785560833923937358906343287416580980640755698830777591985243173236001282446183472413949183562675699871261905669692479191835626756998712619056696924791, 65537, 552140211553831259986200838136425907433611805962565103791847110268943833335364984134359651806590449030993117212443730498321325050674206465403470193504443853999997959213570875513466545102825682592329790042169677742810105093234698344824927012425271868633917262304505456721028831044684193107077013940690818347446323433631045175006084720923318769852782559730392665352113762590757738178634972988165869531593839872247619547119793040753165115979535468508317163432603641962065027435687965465992240566026789692657953351983734123703914994222590398053039810477072185949720202822148209311990929978823769868836029728904990336807341592478849392877497121014943360677190300611057635549624580655929356599770262370228514703882508340022174434978449022210156537749069513202619295413299691770839426849880721786973495777244530583594208659804874613365284467894050997374403458570509317884014855272146701086369341721849094251066694646123646053369712116525832003344653514011153743469275894688563301874236180687116945045828356637412718573111522470228050959640043780426825883730075570503862512103589216040160341912579713624104896992738752581460415766997686361090129522155300302654461867216971914358511893323541978100)
```

```
Cyphertext: b'*\x92\xecn\x88\xd9!~\x9d\xdcRJ"\xa1/\xf6m\xde`\xa0\x18Y*{Zj\xe3\x89\xb1
\xe0\x13T?\xd4\xf4d\x8f\xf2\xfd\xdf&y\xdd\xda/\xa2\x1d\xa2N\xf1\xfc\xa9\xd6?\xe0@\xf1
\x01\x15\xb7\xc4O\xb9\x8f\x0bF1\xf9\x91s\xbf\xf4\x1f\xa1\x85\xb7J\xe2\xd8U\xa4\xf0\x8
6\xed\xbc\xdc]m\xb8Y\xf5c\xf6eB3\x00\xd8\x9d\xcd\x9d\xddC \x18\xc0\x1c\xdfY\xd2\xfe\x
81# A\xb2V\x07\xb6\x0f\x86\xc3\x82\xd8\x93\xcd\xd2\xeab\x8f\xc0\xcf\x08\xff\x16BH_\xc
0T\xb8\x89V\x97mq*\xa6\x88\x91\x7f;\x95=\xcf\x92\'\xe2\xe7\x98e\xf04\x98E,\x8d\xa0\x9
0\xc4\xdfN5\xf0\xe6\r9@B<\xda\xc2K\x11p\x1e\x8d*\x08\x7f\x1d\xd4\x01K\x96\xae\xdd\xc8
\xfe\xaa\xcd\xa08\xb6\x8cDT\xd9p\xd6\xf5\xa9\\\x02\xa32N\x99\x91\x0f\xaf\xc7\xdc\x89
\xf7A|\xd5\x0fT)X\xa2\xa8\x17\x96\xbe2W"\xc0\xfcZ_\xab\xebD\x98E!L\xd5\x90\xbf\xbd?'
Hello, World!
```

```
Signature: b'\x17on[\x08\xdeX\x7fY4\xb41\xee/\xb3v-\xc8U6?\xcbV(Ak\xf9\xe6]\x19\x13\x
fb\xff\x86\xc3\xca.\xa7\x83\xb8\x03$\xe0\xdc\xa7M0\xee\x8fS\xf7\xc6\t\xbc\x88\xb9CRKA
\x01\xba\x0b&}\x13.\xcb\xc9|)\xe5/%\x17\xdc\x0f\xa6\x04M\x01\xa3\x89\xe1\xfe/\x98QB\x
eaC\x0e\xe5\xa6\xb0%\xa9BU \xf9\xd8\x82\xfb\xd8\x00q\xcf"\x0e]\xf6U\x08C-/\xcb\xa5G\x
e9\x03-y\xebq\x9b\xb7\x90\x82\x12\xc7\xac\x99\xf1X`\xcb\xd1]\rJE(\x979<t\x16\xfd2\x80
\x8d\xec\xe5\x8e\xc8\xf5K\xd5\xfa\xb9b\xf8x\xce)TB\x15\xdd\xd6"\x16(+\x86f\xa9\xcb\xc
aY\xf5.\xe3\xcbJ_\x97\x01FV\xe7\xc4\xe8S\xa5\x9f\xdd\xce0\x00\x1a\x1bh\x94j\x12\xce
$\x0e0vm\xe30,|\xbe\xdc6!\x172\xcd\xa4\x99QX\xdd\x86Q\xf7\x16a\'\x14\xed+\x12\xa1\xb9
G\xe9~[\xa11\x89\xac2\x9d\xf8\xf4\xb6\xc6'
Verified as SHA-256
Invalid signature
```

Here's the improved Person - not a subclass, since it redefines everything

In [15]:
```
import rsa

class SecurestPerson :
    def __init__(self):
        (self._pubkey, self._privkey) = rsa.newkeys(2048)

    def getPublic(self):
        return self._pubkey

    def decrypt(self, message):
        return rsa.decrypt(message,self._privkey)

    def signString(self,string) :
        return string, rsa.sign(string, self._privkey, 'SHA-256')
```

That's now the Satoshi example again

Slightly recoded to fit the RSA API

In [16]:
```
import rsa

# Satoshi
satoshi = SecurestPerson()
satoshiKey = satoshi.getPublic()
print("I know the public key of Satoshi: ",satoshiKey)

# Craig
craig = SecurestPerson()

# Secret message
secret = "1234".encode('utf8')

# Use P1's public key to encrypt the secret
challenge = rsa.encrypt(secret, satoshiKey)
print("Challenging with",challenge)

# Let both decrypt it ==> We know who is Satoshi (the one answering with the secret)
try:
    print(satoshi.decrypt(challenge).decode('utf8'))
except rsa.DecryptionError :
    print("Satoshi cannot decrypt!")
try:
    print(craig.decrypt(challenge).decode('utf8'))
except rsa.DecryptionError :
    print("Craig cannot decrypt!")
```

```
I know the public key of Satoshi:  PublicKey(164020146778627666887985801476414881475057293453008409734908405967008871609566626904198848667330588521629366401117195023653232702212536557860742393291570677359435230891821691861988369747755670593527411443461719437290101123871239277792318690062509668002761747693249100085770418023321176083399046155238936527502741761213480343022571269778401896655669359936250345660728903109067412237457294434614933368504118036674777061565646983346265490410928427948716333741653447065811991119419388209889698849100562984666718277923496732651572825205896470008306964815215658544718702275967809978051419511772701696296984292683850991389431, 65537)
Challenging with b"n\xcal\x00\xd8 Pk^\x03\\O\x01\x95\xc9\xbdS\xe6\xb1\x9ajH,\xce\x9cm\x880\xdbzd\x86\xbbF#Ym\xbf`\xf3l\xe5\xc8\xdd0h\xf6g\xf7\xe8\xaa\xca\xbfak\xa7+\xc1\x04\xefU\xb9\xf27V\xbd\xc5X\x95\xd6\x11\xfe\x84'\x14/Oq(Ju;\xcf\xf4CT\x9e\x95\x8f]0\x0b\x8a\x00\xbc{\x90\xeeg\xf2\xc9\x1cH\xb1\x1b\xa3\xdd\x01V\xd9\x13\xfb$?\x918\x97\xa6\xd3%\xc5\x94\xb8K\x83\x04T\x1b9\x10)\x08*5r\xa2&\xc18\xa5\xcdC\x0f\xa9\xa9{\t\x14\xe9}\xd3\xdd\xcf\x19\xc3D\xb5\xd0\xd5\x12\xb1\xe1\xe6\xcc\xbd;\xf0R\x86v\x1a\x9b\xe8\xbf\x13#\xa5\xdfs\x01\xf7\x82G\xd4\x0f:D\xae]5c\x9e\x8fI>,#\xea\xd1\xef\x06;\x19\x88C\xf8\xa7\xb8\x80\x80\xfes\x11!\xc1\xaf\xd3\x1a\xff\x8eS\x04\x17\xf2P\x95Z3\xae nI\xfd\x9c\xef\x8c\xfa\xd5\xbfB\x9e\xaaZ\xfd\x89\xf1i+B\xd5L:\x8b\xa6\xda\xa0"
1234
Craig cannot decrypt!
```