



Python for Security

Cryptography Advanced Topics

by

Dr. Mona Fouad

mfouad@nti.sci.eg

2021-2022

Course Outline



- ✓ First Day—Python Crash Course
 - ✓ Programming Overview
 - ✓ Python Installations
 - ✓ Python Programming Concepts and Hands-on

Course Outline (cont.)



- ✓ Second Day—Cryptography using Python
 - ✓ Cryptography Overview
 - ✓ Caesar Ciphering
 - ✓ Modular Arithmetic
 - ✓ Modular Inverse
 - ✓ Greatest Common Divisor
 - ✓ Affine Ciphering
 - ✓ Attacking Encrypted Messages

Course Outline (cont.)



- Third Day—Advanced Topics
 - Vigenère and one-time pad ciphers
 - Base64 CODEC
 - Hashing and password verification
 - DES and AES Cryptography
 - Public Key and Digital Signature
- References

Python Review



- Receive the Assessment Google Form, solve questions and submit, see your score.

Contents



- Vigenère and One-Time Pad Ciphers
- base64 CODEC
- Hashing (MD5, SHA-x)
- cryptography Module
- crypto Module
- hashlib Module
 - DES and AES Cryptography
 - Public Key and Digital Signature

Vigenère Cipher



- Multiple key cipher
- The original key is split into individual sub-keys.
- Instead of encrypting the whole plaintext with one Caesar key, we apply a different Caesar key to each letter of the plain-text.
- Example—if we use a Vigenère key of PIZZA, the 1st subkey is P, the 2nd subkey is I, the third and 4th subkeys are both Z, and the 5th subkey is A. The 1st subkey encrypts the first letter of the plaintext, the 2nd subkey encrypts the second letter, and so on. When we get to the sixth letter of the plaintext, we return to the 1st subkey.

Vigenère Cipher (cont.)

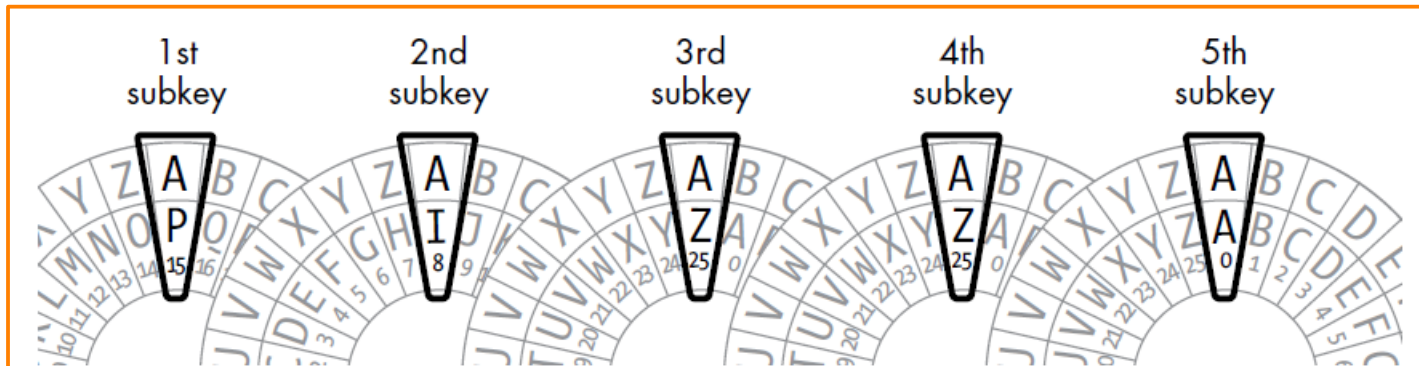


Figure 18-1: Multiple Caesar ciphers combine to make the Vigenère cipher

Each subkey is converted into an integer and serves as a Caesar cipher key. For example, the letter A corresponds to the Caesar cipher key 0. The letter B corresponds to key 1, and so on up to Z for key 25,

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
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

Vigenère Cipher (cont.)

Encrypting Letters with Vigenère Subkeys

plaintext
COMMON
SENSE IS
NOT SO
COMMON

Plaintext letter	Subkey	Ciphertext letter	Plaintext letter	Subkey	Ciphertext letter
C (2)	P (15)	R (17)	S (18)	Z (25)	R (17)
O (14)	I (8)	W (22)	N (13)	Z (25)	M (12)
M (12)	Z (25)	L (11)	O (14)	A (0)	O (14)
M (12)	Z (25)	L (11)	T (19)	P (15)	I (8)
O (14)	A (0)	O (14)	S (18)	I (8)	A (0)
N (13)	P (15)	C (2)	O (14)	Z (25)	N (13)
S (18)	I (8)	A (0)	C (2)	Z (25)	B (1)
E (4)	Z (25)	D (3)	O (14)	A (0)	O (14)
N (13)	Z (25)	M (12)	M (12)	P (15)	B (1)
S (18)	A (0)	S (18)	M (12)	I (8)	U (20)
E (4)	P (15)	T (19)	O (14)	Z (25)	N (13)
I (8)	I (8)	Q (16)	N (13)	Z (25)	M (12)

ciphertext
RWLLOC
ADMST QR
MOI AN
BOBUNM

Choose a key that could not be discovered by dictionary attack...

Vigenère Cipher (cont.)



```
conda install -c conda-forge pyperclip  
or  
python3 -m pip install pyperclip  
or  
Pip install pyperclip
```

vigenereCipher.py,

- pyperclip.py

Vigenère Cipher (cont.)



vigenereCipher.py

- Setting Up Modules, Constants, and the main() Function
- Building Strings with the List-Append-Join Process
- Encrypting and Decrypting the Message
- Calling the main() Function

Hacking Vigenère Cipher



- A ***brute-force dictionary attack*** tries every word in the dictionary file as the Vigenère key, which works only if the key is an English word, such as RAVEN or DESK.

[vigenereDictionaryHacker.py](#),

- [detectEnglish.py](#),
- [vigenereCipher.py](#),
- [pyperclip.py](#)

Vigenère Dictionary Hacking Program



[vigenereDictionaryHacker.py](#)

- *Kasiski examination*—is a process that we can use to determine the length of *the Vigenère key used to encrypt a ciphertext*.
- Finding Repeated Sequences
- Getting Factors of Spacings
- Getting Every n^{th} Letters from a String
- Using Frequency Analysis to Break Each Subkey
- Brute-Forcing Through the Possible Keys

One-Time Pad Cipher



- The one-time pad cipher is a Vigenère cipher that becomes unbreakable when the key meets the following criteria:
 - 1) It is exactly as long as the encrypted message.
 - 2) It is made up of truly random symbols.
 - 3) It is used only once and never again for any other message.

One-Time Pad Cipher (cont.)



- Usually, a large list of one-time pad keys is generated and shared in person, and the keys are marked for specific dates (for example).
- In details, if we received a message from our collaborator on October 31, we would just look through the list of one-time pads to find the corresponding key for that day.
- Frequency analysis attack could hack the one-time pad cipher.
- But, if the key is the same length as the message, each plaintext letter's subkey is unique, meaning that each plaintext letter could be encrypted to any ciphertext letter with equal probability.
- For example, to encrypt the message IF YOU WANT TO SURVIVE OUT HERE, YOU'VE GOT TO KNOW WHERE YOUR TOWEL IS, we remove the spaces and punctuation to get a message that has 55 letters.

One-Time Pad Cipher (cont.)

Different keys may produce the same encrypted text!!

Plaintext	IFYOUWANTTOSURVIVEOUTHEREYOUVEGOTTOKNOWWHEREYOURTOWELIS
Key	KCQYZHEPXAUTIQEKXEJMORETZHZTRWWQDYLBTTEJMEDBSANYBPXQIK
Ciphertext	SHOMTDECQTILCHZSSIXGHYIKDFNNMACEWRZLGHRAQQVHZGUERPLBBQC

-1: Encrypting an example message using a one-time pad

Plaintext	THEMYTHOFOSIRISWASOFIMPORTANCEINANCIENTEGYPTIANRELIGION
Key	ZAKAVKXOLFQDLZHWSQJBZMTWMMNAKWURWEXDCUYWKSGORGHNNEDVTCF
Ciphertext	SHOMTDECQTILCHZSSIXGHYIKDFNNMACEWRZLGHRAQQVHZGUERPLBBQC

-2: Encrypting a different example message using a different key but producing the same ciphertext as before

Avoiding the Two-Time Pad



- A two-time pad cipher refers to using the same one-time pad key to encrypt two different messages. This creates a weakness in an encryption, you're giving crucial information to the hacker.

base64 CODEC



- base64 encoding converts the binary data into text format, which is passed through communication channel where a user can handle text safely.
- base64 is also called as **Privacy enhanced Electronic mail (PEM)** and is primarily used in email encryption process.
- Python includes a module called base64 which includes two primary functions as given below:
 - `base64.encode(input, output)` – It encodes the input value parameter specified and stores the decoded output as an object.
 - `base64.decode(input, output)` - It decodes the input value parameter specified and stores the decoded output as an object.

base64 Module



```
import base64
# base64 Encoding
encoded_data = base64.b64encode(b"Encode this text")
print("Encoded text with base 64 is")
print(encoded_data)
```

```
Encoded text with base 64 is
b'RW5jb2RlIHRobXMgdGV4dA=='
```

```
# base64 Decoding
decoded_data = base64.b64decode("RW5jb2RlIHRobXMgdGV4dA==")
print("decoded text is ")
print(decoded_data.decode())
```

```
decoded text is
Encode this text
```

base64 vs ASCII



- You can observe the following differences when you work on ASCII and base64 for encoding data:
 - When you encode text in ASCII, you start with a text string and convert it to a sequence of bytes.
 - When you encode data in Base64, you start with a sequence of bytes and convert it to a text string.
- **Drawbacks**
 - Base64 algorithm is usually used to store passwords in database.
 - Each decoded word can be encoded easily through any online tool and intruders can easily get the information.

Hashing



- Hashing is the process used for converting a given key into another value.
- Hashing has two main purposes:
 - to put a fingerprint on a file so you can tell whether it has been altered (Authenticating), and
 - to conceal passwords so you can still recognize the correct password and enable login but a person who steals the hash cannot easily recover the password from it.
- A hash function is used to generate the new value according to a mathematical algorithm.
- The result of a hash function is known as a hash value or simply, a hash.
- hashlib module is an implementation of hashing in Python.

hashlib Module

`pip install hashlib`



- hashlib is a hashing module in Python
- A very common hash is the Message-Digest algorithm (MD5). It is broken but still used!!!
- It's 128 bits long, which is rather short for a hash function, and it's reliable enough for most purposes.
- The Secure Hash Algorithms (SHA) was designed to be an improvement on MD5.
- Both SHA-2 and SHA-3 have various lengths, but the most common lengths are 256 and 512 bits.
- Other SHA-n are now available.

MD5 and SHA Hashes

```
import hashlib
```

```
txt="HELLO"  
txt_sha1=hashlib.new("sha1",txt.encode()).hexdigest()  
txt_sha2=hashlib.new("sha256",txt.encode()).hexdigest()  
txt_sha3=hashlib.new("sha512",txt.encode()).hexdigest()
```

```
print(txt_sha1)  
print(txt_sha2)  
print(txt_sha3)
```

```
c65f99f8c5376adadddc46d5cbcf5762f9e55eb7  
3733cd977ff8eb18b987357e22ced99f46097f31ecb239e878ae63760e83e4d5  
33df2dcc31d35e7bc2568bebf5d73a1e43a0e624b651ba5ef3157bbfb728446674a231b8b6e97fa1e570c3b1de6d6c677541b262ac22afda58  
78fa2b591c7f08
```

```
txt_md5=hashlib.new("md5",txt.encode()).hexdigest()  
txt_md5
```

```
'eb61eead90e3b899c6bcbe27ac581660'
```

```
txt="HELLO!"  
txt_md5=hashlib.new("md5",txt.encode()).hexdigest()  
txt_md5
```

```
'9ac96c64417b5976a58839eceaa77956'
```


Password Verification



- Hashing procedure for password verification
 - Receive a password from the user.
 - Hash the entered password.
 - Verify the password for authentication purpose.

Password Verification (cont.)

```
import uuid
import hashlib

def hash_password(password):
    # uuid is used to generate a random number of the specified password
    salt = uuid.uuid4().hex
    return hashlib.sha256(salt.encode() + password.encode()).hexdigest() + ':' + salt

def check_password(hashed_password, user_password):
    password, salt = hashed_password.split(':')
    return password == hashlib.sha256(salt.encode() + user_password.encode()).hexdigest()

new_pass = input('Please enter a password: ')
hashed_password = hash_password(new_pass)
print('The string to store in the db is: ' + hashed_password)
old_pass = input('Now please enter the password again to check: ')

if check_password(hashed_password, old_pass):
    print('You entered the right password')
else:
    print('Passwords do not match')
```

```
Please enter a password: Hello!
The string to store in the db is: eddc2c282e27924b85cfe929bd6ec70f9e516b0a395eb6688ec9a8f6a
45fc
Now please enter the password again to check: hello1
Passwords do not match
```

Cracking hashes with wordlists



- You just use the same procedure. Make a series of guesses, hash them, and hunt for your answer.

```
for c in ('6799'):
    p="P@sw0rd99"+c
    h=hashlib.new("md5",p.encode()).hexdigest()
    print(p,h)
```

```
P@sw0rd996 2d97149df8c642e162011ff44d8b9ad5
P@sw0rd997 b712c32f140b7d0dd9ebaab23f43c9e5
P@sw0rd999 9a03df6d1dc648fe0f5b96cdcad8b89e
P@sw0rd999 9a03df6d1dc648fe0f5b96cdcad8b89e
```

```
hashlib.new("md5", "P@sw0rd999".encode()).hexdigest()

'9a03df6d1dc648fe0f5b96cdcad8b89e'
```

Cryptography Module

pip install cryptography

- Generate key, encrypt a text, decrypt the encrypted text

```
from cryptography.fernet import Fernet
key = Fernet.generate_key()
cipher_suite = Fernet(key)
cipher_text = cipher_suite.encrypt(b"This example is used to demonstrate cryptography module")
plain_text = cipher_suite.decrypt(cipher_text)
```

```
print(cipher_text.decode())
print()
print(plain_text.decode())
```

```
gAAAAABjUuWhOkNGeEEgqfI3rnU0XFcqCWklGJWusNbCzNJUQPvo_ZpmHE8QFkokF7RkoHDSHsoH_p1FF0mCY8rmJRDdhZ0YBH1huuLKZPNZQEaose2d4bocZ3NU9RM
r7Kn05R1H2YVL44oiM4impUSyjoYj8jz1Hg==
```

This example is used to demonstrate cryptography module

Block Ciphers

DES and AES



- DES is the Data Encryption Standard.
- AES is the Advanced Encryption Standard.
- Both DES and AES are block based ciphers.
- DES block size is of 64 bits length with a key of size 56 bits.
- AES block size is of 128-bits long
- There are three key sizes for AES: 128, 192, and 256-bits.
- The most common type of AES is the 128-bit key size.

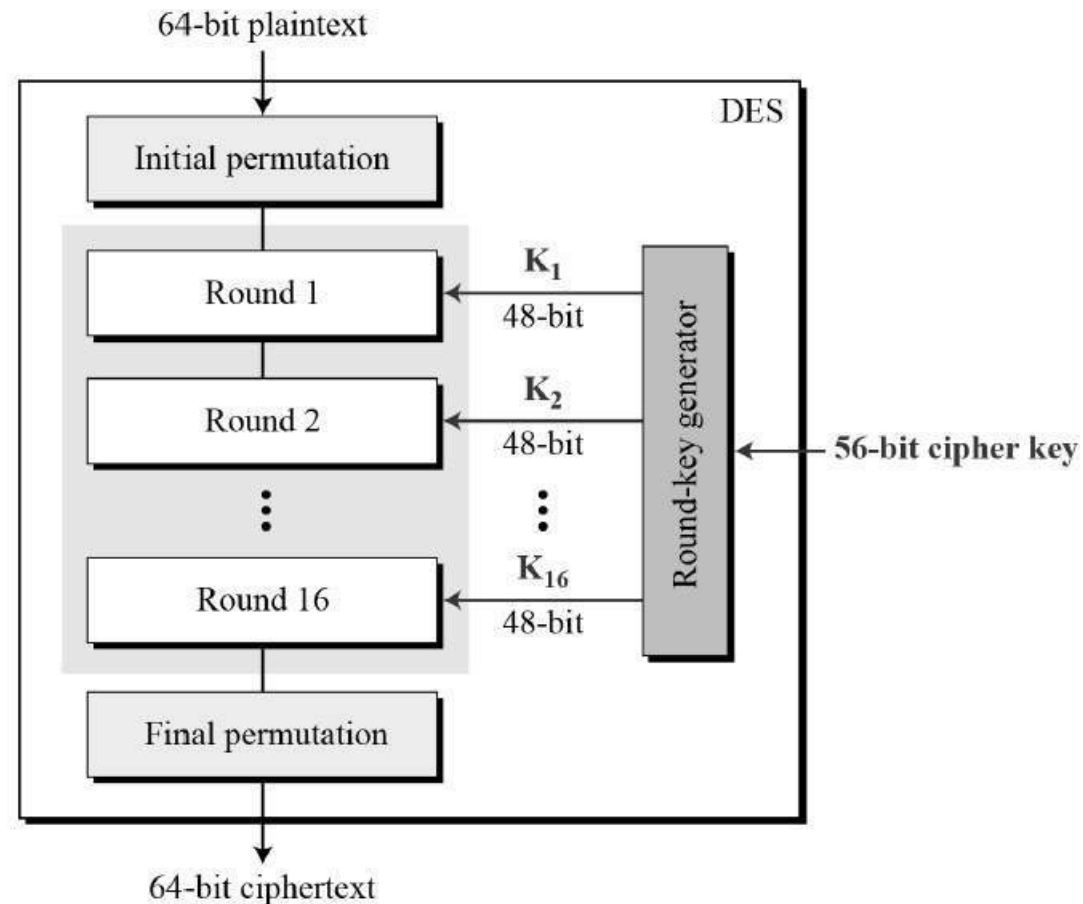
	DES	AES
Developed	1977	2000
Key Length	56 bits	128, 192, or 256 bits
Cipher Type	Symmetric block cipher	Symmetric block cipher
Block Size	64 bits	128 bits
Security	Proven inadequate	Considered secure

DES



- Data Encryption Standard (DES)
 - Confusion—obscured relationship between plain and cipher texts.
 - Diffusion—spreading the influence of each plain-text bit over several cipher-text bits. (ex. permutation)

DES Algorithm Description



https://www.tutorialspoint.com/cryptography/data_encryption_standard.htm

DES Algorithm Description



- DES takes 64-bit plain text and turns it into a 64-bit cipher-text.
- The key is only 56 bits, and the other 8 bits are used for parity check.
- The process begins with the 64-bit plain text block getting handed over to an initial permutation (IP) function.
- The initial permutation (IP) is then performed on the plain text.
- Next, the initial permutation (IP) creates two halves of the permuted block, referred to as Left Plain Text (LPT) and Right Plain Text (RPT).
- Each LPT and RPT goes through 16 rounds of the encryption process.
- Finally, the LPT and RPT are rejoined, and a Final Permutation (FP) is performed on the newly combined block.
- The result of this process produces the desired 64-bit ciphertext.

<https://www.simplilearn.com/what-is-des-article>

Crypto Module

python3 -m pip install pycryptodome
or
conda install pycryptodome



- DES Example

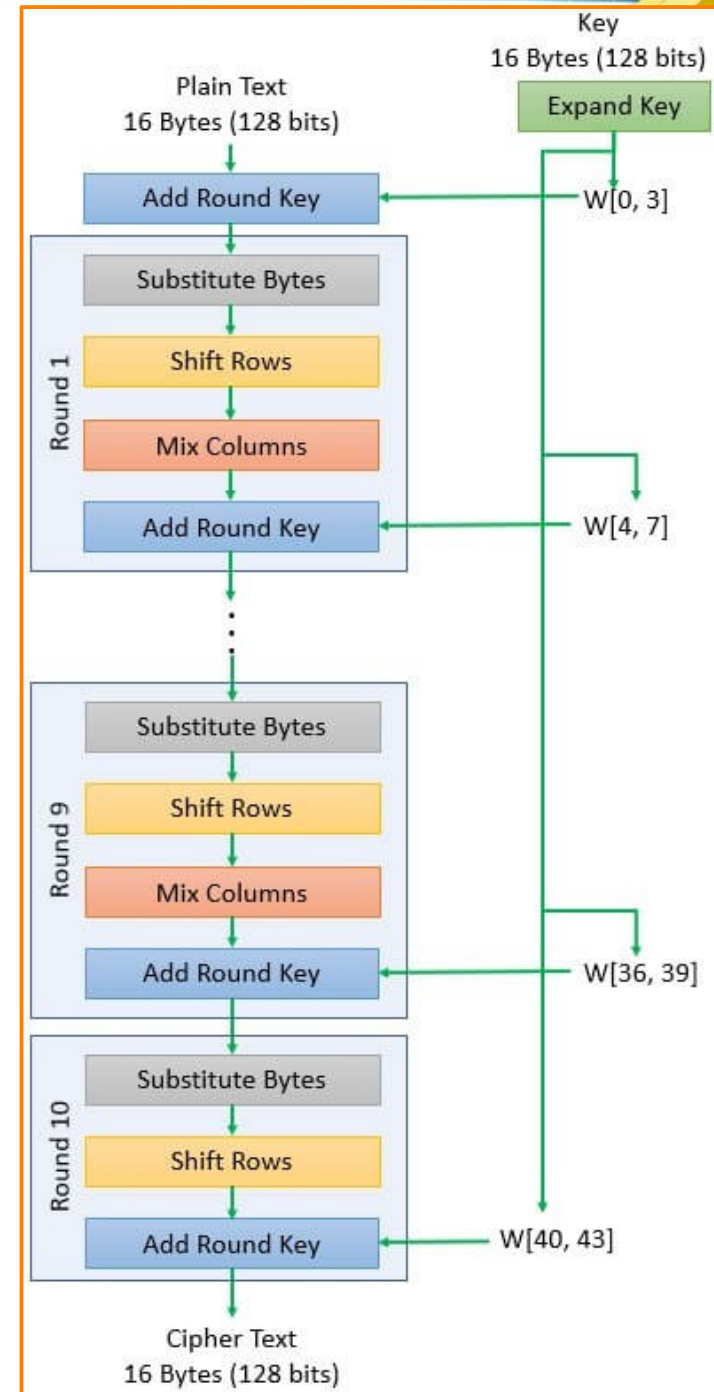
```
1 from Crypto.Cipher import DES
2 from Crypto import Random
3 key = "mysecret"
4 des1 = DES.new(key)
5 des2 = DES.new(key)
6
7 text = '!!Good Morning!!'
8 print("Original message: ", text)
9 cipher_text = des1.encrypt(text)
10 print("Encrypted message: ", cipher_text)
11 dtext = des2.decrypt(cipher_text)
12 print("Decrypted Original Message: ", dtext.decode())
13
```

Original message: !!Good Morning!!

Encrypted message: b'\xb5\xb9@\xeauH&\x93\xbf\xcc\xbc\xbc\xe6\x06\xe2?'

Decrypted Original Message: !!Good Morning!!

AES




AES Algorithm Description



- **Key expansion**, which creates new rounded keys, for each subsequent round of encryption.
- **Round key addition**, during which the initial round key is added to the mix of data that has been divided.
- **Byte substitution**, which substitutes every byte with a different byte based on the Rijndael S-box substitution box.
- **Row shifting**, which moves every row of the divided data one space to the left for the second row, two spaces to the left for the third row, and three spaces to the left for the fourth row.
- **Column mixing**, which uses a pre-established matrix to multiply the divided data's columns and create a new block of code.
- **Round key addition**, during which another round key is added to the mixture of columns.

<https://www.trentonsystems.com/blog/aes-encryption-your-faqs-answered>

AES Example



```
1 from Crypto.Cipher import AES
2 from Crypto.Random import get_random_bytes
3 key = "Sixteen byte key"
4 data = "Secret: 16 bytes"
5 cipher = AES.new(key)
6 data_enc = cipher.encrypt(data)
7 print(data_enc)
8 data_dec = cipher.decrypt(data_dec)
9 print(data_dec.decode())
```

```
b'et\xc3\x9aS\xaaUF\xdd\x18"\xd5\xd9;\x14]'
Secret: 16 bytes
```

Public Key Cipher



- Asymmetric Cryptography
- Encryption and decryption are performed using different keys, namely; public key, and private key, respectively.
- A message encrypted using the encryption key (public key) can only be decrypted using the decryption key (private key).
- If someone obtains (stole) the encryption key, they won't be able to read the original message because the encryption key can't decrypt the message, but s(he) can hide the original message and send an altered one...

Possible Attacks

- Brute-Force (try all possible keys)
- Cryptanalysis (detect repetitive patterns)
- Man-In-The-Middle (MITM)—a hacker could catch a message, replace it, encrypt it using the public key (stole before), and then send the altered message instead...




RSA Algorithm



- Generate public and private keys
- Encryption process
 - Partition the text message to be encrypted into blocks.
 - Apply the public key to each block
- Decryption process
 - Apply the private key to each received block.
 - Assemble decoded blocks together to retrieve the original message.

The maximum block size depends on the symbol set size and key size. The equation $2^{\text{key size}} > \text{symbol set size} \times \text{block size}$ must hold true.

Generating Public and Private Keys



- The public key will be the two numbers n and e .
- The private key will be the two numbers n and d .
- The three steps to create these numbers are as follows:
 - 1) Create two random, distinct, very large prime numbers: p and q .
$$n = p \times q$$
 - 2) Create a random number, called e , which is relatively prime to $(p - 1) \times (q - 1)$
 - 3) Calculate d that is the *modular inverse* of e .
- The d number must be kept secret because it can decrypt messages.

RSA Encryption and Decryption



- The general equations for the RSA encryption and decryption:
 - Encryption: $C = M^e \bmod n$
 - Decryption: $M = C^d \bmod n$
- The public key is (n, e)
- The private key is (n, d)
- e is relatively prime with the number $(p - 1) \times (q - 1)$
- d is the modular inverse of e and $(p - 1) \times (q - 1)$.

Python's `pow()` function uses a mathematical trick called modular exponentiation to quickly calculate such a large exponent.

Modular Inverse



- If modular operator is used for encryption, modular inverse should be used for decryption.
- Modular inverse ***d*** for two numbers ***e***, ***n*** is
$$(e * d) \% n = 1$$
- Modular inverse is a brute-force process...

Applying the RSA Algorithm



- `makePublicPrivateKeys.py`
 - `primeNum.py` and
 - `cryptomath.py` modules
- `publicKeyCipher.ipynb`

Assignment: add a cell to call decryption for the encrypted message.

RSA keys-generation



```
1 from Crypto.PublicKey import RSA
```

```
1 from Crypto.PublicKey import RSA
2 from Crypto.Cipher import PKCS1_OAEP
3 import binascii
4
5 keyPair = RSA.generate(3072)
6
7 pubKey = keyPair.publickey()
8 print(f"Public key: (n={hex(pubKey.n)}, e={hex(pubKey.e)})")
9 pubKeyPEM = pubKey.exportKey()
10 print(pubKeyPEM.decode('ascii'))
11
12 print(f"Private key: (n={hex(pubKey.n)}, d={hex(keyPair.d)})")
13 privKeyPEM = keyPair.exportKey()
14 print(privKeyPEM.decode('ascii'))
```


RSA Cipher in Python



- Encrypting a message using the generated RSA public key:

```
1 from Crypto.Cipher import PKCS1_OAEP
2 import binascii
3
4 msg = b'A message for encryption'
5 encryptor = PKCS1_OAEP.new(pubKey)
6 encrypted = encryptor.encrypt(msg)
7 print("Encrypted:", binascii.hexlify(encrypted))
```

- Decrypting the encrypted message using the RSA private key:

```
1 decryptor = PKCS1_OAEP.new(keyPair)
2 decrypted = decryptor.decrypt(encrypted)
3 print('Decrypted:', decrypted.decode())
```

- see `cryptoPublicKey_ex.ipynb`

Authentication Problem



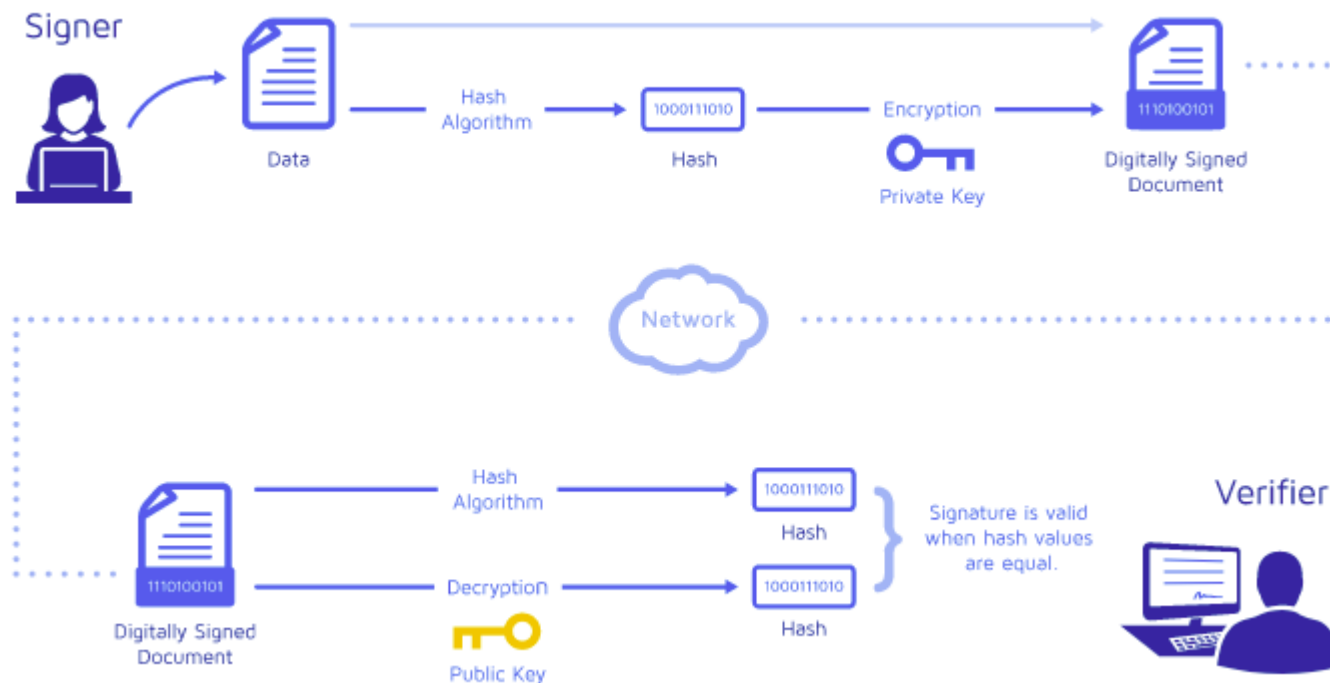
- Using cryptography to secure transmitted messages ensure confidentiality, but not authenticity.
- How could you trust that the message you decrypt is the original sent message!!!

Digital Signature



- Digital signatures are like electronic “fingerprints.” In the form of a coded message, the digital signature securely associates a signer with a document in a recorded transaction.
- Digital signatures use a standard, accepted format, called Public Key Infrastructure (PKI), to provide the highest levels of security and universal acceptance.

Digital Signature (cont.)



<https://www.docusign.com/how-it-works/electronic-signature/digital-signature/digital-signature-faq>

Note: You can also encrypt the message itself, which provides the confidentiality of the information in the message.

Digital Signature (cont.)



- Digital signatures are part of integrity and authentication services. It provides proof of origin.
- Only the sender knows the private key, which provides strong evidence that the sender is the originator of the message.
- Steps:
 - 1) The sender computes a message digest (hashing) and then encrypts the digest using the sender's private key, forming the digital signature.
 - 2) The sender transmits the digital signature with the message.
 - 3) The receiver decrypts the digital signature using the sender's public key, regenerating the sender's message digest.
 - 4) The receiver computes a message digest from the message data received and verifies that the two digests are the same.

Public Key for Message Signature



- see “create_verify_Signature.ipynb”

1) Create keyPairs: Public and private

```
1 from Crypto.PublicKey import RSA
2
3 keyPair = RSA.generate(1024) # generate key pairs
4
5 pubKey = keyPair.publickey()
6 print(f"Public key: (n={hex(pubKey.n)}, e={hex(pubKey.e)})")
7 # PEM: Private Enhanced Message format (simple readable format)
8 pubKeyPEM = pubKey.exportKey()
9 print(pubKeyPEM.decode('ascii'))
10
11 print(f"Private key: (n={hex(pubKey.n)}, d={hex(keyPair.d)})")
12 privKeyPEM = keyPair.exportKey()
13 print(privKeyPEM.decode('ascii'))
```

Public Key for Message Signature (cont.)



- see “create_verify_Signature.ipynb”
 - 2) Hashing the message
 - 3) Generate a signature
 - 4) Send the message and the signature

```
1 import Crypto.Signature
2 from Crypto.Signature import pkcs1_15
3 from Crypto.Hash import MD5
4
5 message = b'Message to be signed'
6 key = RSA.import_key(privKeyPEM)
7 h = MD5.new()
8 h.update(message)
9 # pkcs1_15.new(): Create a signature object for creating
10 # or verifying PKCS#1 v1.5 signatures.
11 signature = pkcs1_15.new(key).sign(h)
```

pkcs: public key codec standard

Public Key for Message Signature (cont.)

- see “create_verify_Signature.ipynb”
 - 5) At the receiver side: hash the message and decrypt the signature, if they are equal then the signature is verified and the message is authenticated.

```
1 key = RSA.import_key(pubKeyPEM)
2 h = MD5.new(message)
3 try:
4     # pkcs1_15.new(): Create a signature object for creating
5     # or verifying PKCS#1 v1.5 signatures.
6     pkcs1_15.new(key).verify(h, signature)
7     print("The signature is valid.")
8 except (ValueError, TypeError):
9     print("The signature is not valid.")
10
```

The signature is valid.

Quiz



- Alice generates a public key and a private key. Unfortunately, she later loses her private key.
 - a) Will other people be able to send her encrypted messages?
 - b) Will she be able to decrypt messages previously sent to her?
 - c) Will she be able to digitally sign documents?
 - d) Will other people be able to verify her previously signed documents?

References



- Anaconda: <https://docs.anaconda.com/anaconda/install/>
- Python Documentation: <https://docs.python.org/>
- Python Packages: <https://pypi.org/>
- Further installations for extra libraries: it is recommended to google search...
- “Cracking Codes with Python.” Copyright © 2018 by Al Sweigart. No Starch Press, Inc. ISBN-10: 1-59327-822-5, ISBN-13: 978-1-59327-822-9
- A Complete Introduction to the Python Language: “Programming in Python 3”, Second Edition, by Mark Summerfield, Copyright c 2010 Pearson Education, Inc., ISBN-13: 978-0-321-68056-3, ISBN-10: 0-321-68056-1



Thank You