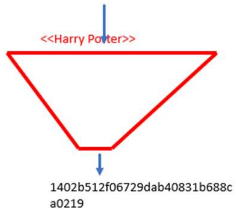
One-way Hash Function

- One-way hash function is a function that maps a **long** input into a **short** output, preserving the security.



Properties of One-way Hash Function

- One-way Hash Properties:
  - One-way: `hash(m) = h`, difficult to find m
  - Collision resistant: Difficult to find m1 and m2 s.t. `hash(m1) = hash(m2)`
- Common One-way Hash Functions:
  - MD series
  - SHA series

MD One-Way Hash Functions

- MD stands for Message Digest
- Developed by Ron Rivest
- Includes MD2, MD4, MD5,and MD6
- Status of Algorithms:
  - MD2, MD4 - severely broken (obsolete)
  - MD5 - collision resistance property broken, one-way property not broken
  - MD6 - developed in response to proposal by NIST

SHA

- Published by NIST
- Includes SHA-0, SHA-1, SHA-2, and SHA-3
- Status of Algorithms:
  - SHA-0: withdrawn due to flaw
  - SHA-1: Collision attack found in 2017
  - SHA-2: Includes SHA-256 and SHA-512; No significant attack found yet
  - SHA-3: Released in 2015; Has different construction structure

One-Way Hash Commands

Linux utility programs

- Example: md5sum, sha224sum, sha256sum, sha384sum and sha512sum

```
$ md5sum file.c
919302e20d3885da126e06ca4cec8e8b   file.c

$ sha256sum file.c
0b2a06a29688...(omitted)...1f04ed41d1   file.c
```

• Hash programming using

Python package PyCryptodome: https://pycryptodome.readthedocs.io/en/latest/src/api.html

```
#!/usr/bin/python3

from Crypto.Hash import SHA512, MD5, SHA224


msg0="Harry Porter"
msg1="Alice in Wonderland"

#hash directly on messages
h512=SHA512.new(msg0.encode())   #.new() initiaze a hash object; i
h224=SHA224.new(msg0.encode())
h5=MD5.new(msg0.encode())

#hash serveral messages sequentially.

mh5=MD5.new()
mh5.update(msg0.encode())   # msg0 is hashed.
mh5.update(msg1.encode())  # now msg0+msg1 is hashed.

print("SHA512({})={}".format(msg0, h512.hexdigest()))
print("SHA224({})={}".format(msg0, h224.hexdigest()))
print("MD5({})={}".format(msg0, h5.hexdigest()))
print("MD5({})={}".format(msg0+msg1, mh5.hexdigest()))
```

Initialize hash object

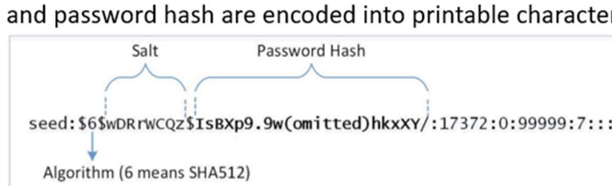
Password Verification

- To login into account, user needs to tell a secret (password)
- Cannot store the secrets in their plaintext
- Solution: Linux stores **hashed** passwords in the /etc/shadow file
- When you provide the password, the system will hash and check the result is identical to the system storage.

```
seed:$6$wDRrWCQz$IsBXp9.9wz9SG(omitted)sbCT7hkxXY/:17372:0:99999:7:::
test:$6$a6ftg3SI$apRiFL.jDCH7S(omitted)jAPXtcB9oC0:17543:0:99999:7:::
```

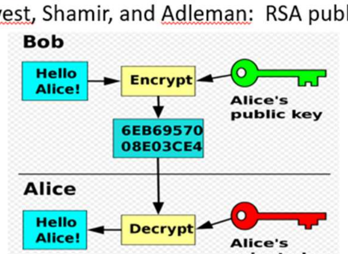
Case Study: Linux Shadow File

- Password field has 3 parts: algorithm used, salt, password hash
- Salt and password hash are encoded into printable characters



Public Key Cryptosystem

- Private Key cryptography: shared key for encryption and decryption
- 1976: Diffie and Hellman: a notion of **public-key encryption**
- 1978: Rivest, Shamir, and Adleman: RSA public key encryption



Modulo Operation

- `a mod n`: the remainder after **a** divided by **n**
- n is called modulus
- Ex. 10 mod 3 = 1 and 15 mod 5 = 0
- **Remember**: +, * and exponentiations under mod n can be done the same way as if no mod n is used.
- **Important**: when you do mod n operation **does not** affect the result.

$$(a+b) \mod n = [(a \mod n) + (b \mod n)] \mod n$$
$$a*b \mod n = [(a \mod n) * (b \mod n)] \mod n$$
$$a^x \mod n = (a \mod n)^x \mod n$$

Modulo Operation: example

- (3+18) mod 7=21 mod 7=0
- (3+18) mod 7=3+ 18mod7=3+4 mod 7=0.
- 3*8 mod 5=24 mod 5 =4 **and** 3* 8 mod 5 =3*3 mod 5=9 mod 5 =4.
- 2^8mod 12=256 mod 12=4
- 2^8 mod 12 =2^4*2^4 mod 12=4*4 mod 12=4

RSA Cryptosystem

We will cover:

- Key generation
- Encryption
- Decryption

RSA: Key Generation

- Need to generate: modulus **n**, public key **e**, private key **d**
- Procedure:
  - Choose large primes p,q
  - n = pq
  - Choose odd e< n (e.g., e=65537)
  - Find d s.t. **ed mod (p-1)(q-1)** = 1 (Euclidean Algoithm)
- Result
  - (e,n) is public key
  - d is private key

ed= k (p-1)(q-1)+1 for some k

RSA Exercise: Small Numbers

- Choose two prime numbers p = 13 and q = 17
- Find e:
  - n = pq = 221
  - (p − 1)(q − 1) = 192
  - choose e = 7
- Find d:
  - ed = 1 mod (p-1)(q-1)
  - 7d = 1 mod 192
- d = 55 (Euclidean algorithm, omitted)

d=inv(e, (p-1)(q-1))

```
def inv(x, m):
    x=x%m
    for y in range(1, m):
        if (x*y%m==1):
            return y
    return 0
```

RSA: Encryption and Decryption

- Encryption
  - treat the plaintext as a number
  - assume M < n
  - $C = M^e \bmod n$
- Decryption
  - $M = C^d \bmod n$

RSA Exercise: Small Numbers (Contd.)

Encrypt M = 36:

C=36^e mod n=**pow**(36, e, n) =179 (in python)

Cipher text (C) = 179 :

M=179^d mod n=**pow**(179, d, n)=36 (in python)

OpenSSL Tools: Generating RSA keys

Example: generate a 1024-bit public/private key pair

- openssl genrsa -aes128 -out private.pem 1024
  - private.pem: Base64 encoding of DER generated binary output

```
$ more private.pem
-----BEGIN RSA PRIVATE KEY-----
MIICWgIBAAKBgQCuXJawrRzJNG9vt2Zqe+/TCT3OxuEKRWkHfE5uZBkLCMgGbYzK
...
mesOrjIfm0ljUNL4VRnrLxrl/1xEBGWedCuCPqeV
-----END RSA PRIVATE KEY-----
```

OpenSSL Tools: Generating RSA keys (Contd.)

Actual content of private.pem

```
$ openssl rsa -in private.pem -noout -text
Enter pass phrase for private.pem:
Private-Key: (1024 bit)
modulus:
    00:c4:5a:9d:8d:f7:ad:0d:e7:60:4e:b3:9c:76:93: ...
publicExponent: 65537 (0x10001)
privateExponent:
    00:a5:86:fe:6b:3f:f0:53:58:4a:88:0e:42:48:74: ...
prime1:
    00:ec:a0:f7:02:8d:79:a0:8b:c5:5b:e6:a0:25:2c: ...
prime2:
    00:d4:6d:9c:4a:35:6b:fb:db:42:20:d8:6e:45:a9: ...
exponent1:
    06:72:d4:88:73:46:8f:43:7f:db:63:4b:95:f7:c4: ...
exponent2:
    00:d1:3c:45:bd:32:71:72:59:bd:00:ed:2d:70:a0: ...
coefficient:
    22:f5:95:05:81:c4:fd:3e:52:99:16:b5:66:92:52: ...
```

OpenSSL Tools: Extracting Public Key

- openssl rsa -in private.pem -pubout > public.pem
- Content of public.pem:

```
-----BEGIN PUBLIC KEY-----
MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDEWp2N960N52BOs5x2k53WglVn
iAv5oUemZdfnGP1qUhTMZfhSbD27eOUJZAEdrMS/4Nax/BJIxz6N+L2K2cQQasJY
Gqf1PetXKtYakzgd5dBuB3aogOTJaBSt8/A0DBK2MtwNMnBxeZWnf4DK8Glsbp2S
nsGmCdceQ4nelGZbIwIDAQAB
-----END PUBLIC KEY-----
```

```
$ openssl rsa -in public.pem -pubin -text -noout
Public-Key: (1024 bit)
Modulus:
    00:af:1a:d9:ca:91:91:6b:b6:d0:1d:56:7a:1b:2d: ...
Exponent: 65537 (0x10001)
```

Programming using Public-Key Crypto APIs

- Python:
  - use Python package PyCryptodome:
    https://pycryptodome.readthedocs.io/en/latest/src/api.html
- We will cover:
  - Encryption and Decryption
  - Digital Signature
  - AES

Public-Key Cryptography APIs: Encryption

- There are different implementations of RSA
- For better security, it is recommended that **OAEP** is used
- Lines in code (example on next slide):
  - Line (1): import the public key from the public-key file
  - Line (2): create a cipher object using the public key

Public-Key Cryptography APIs: Encryption

encrypt_RSA.py

```
#!/usr/bin/python3

from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

message=b'This is my message\n'

key=RSA.import_key(open('Test.pub').read())

cipher=PKCS1_OAEP.new(key)

ciphertext=cipher.encrypt(message)

f=open('ciphertext.bin', 'wb')

f.write(ciphertext)

f.close()
```

- RSA encrypt/decrypt function
- RSA key function
- Convert key in byte string to key object
- Initialize cipher object (with key)
- cipher object has encrypt/decrypt functions

Public-Key Cryptography APIs: Decryption

Uses the private key and the decrypt() API

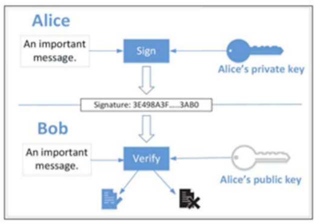
decrypt_RSA.py

```
#!/usr/bin/python3

from Crypto.Cipher import PKCS1_OAEP
from Crypto.PublicKey import RSA

ciphertext = open('ciphertext.bin', 'rb').read()

key_str = open('Test.key').read()
prikey = RSA.import_key(key_str, passphrase='dees')
cipher = PKCS1_OAEP.new(prikey)
message = cipher.decrypt(ciphertext)
print(message)
```

Convert key string to key object: need password

Digital Signature

- Goal: provide an authenticity proof by signing digital documents
- Idea: private key to sign and every body can verify using public key.



Digital Signature using RSA

- The RSA signature for message m can be defined as

  Digital signature $s = m^d \bmod n$

  Verification: $s^e = m \bmod n$?

  why correct: $s^e = m^{de} = C^d = m \bmod n$ (RSA decryption)

- However, m and m+n has the same signature, not good!
- Actual signature: $s = H(m)^d \bmod n$ for a hash function H.
- Verification: $\mathbf{s^e = H(m) \bmod n}$?
- H can be MD5, Sha256, Sha512, etc.

Python Digital Signature using PSS: sign

sign_RSA.py

```
#!/usr/bin/python3

from Crypto.Signature import pss
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA

message=b'I owe you $3000'
key_str=open("private.pem").read()
key=RSA.import_key(key_str, passphrase="dees")
h=SHA256.new(message)
print(h.hexdigest())
signer=pss.new(key)
sig=signer.sign(h)
open("signature1.bin", "wb").write(sig)
```

- Signature function
- Signature object
- Signature object has sign/verify functions

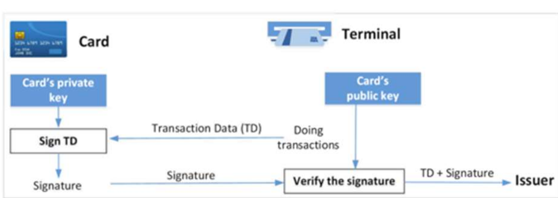
Python Digital Signature using PSS: verify

verify_RSA.p

```
#!/usr/bin/python3

from Crypto.Signature import pss
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA

message = b'This is my message!\n'
signature= open('signature.bin', 'rb').read()
key = RSA.import_key(open('public.pem').read())
h = SHA256.new(message)
verifier = pss.new(key)

try:
    verifier.verify(h, signature)
    print("The signature is valid.")
except (ValueError, TypeError):
    print("The signature is NOT valid.")
```

Applications: Credit Card Authentication

- Credit card company needs to know if the transaction is authentic
- Transaction needs to be signed by the card using its private key
- Verified Signature:
  - To issuers: card owner has approved the transaction
  - To honest vendor: enables the vendor to save the transactions and submit them to credit card company later



Hybrid Encryption

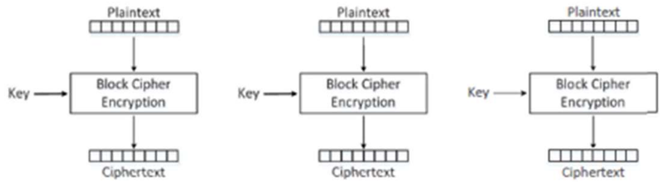
- High computation cost of public-key encryption
- Public key algorithms used to exchange a *secret session key*
- Key (content-encryption key) used to encrypt data using a symmetric-key algorithm. Example:

  $K^e \bmod n$, AES_k(b'This is my secret')

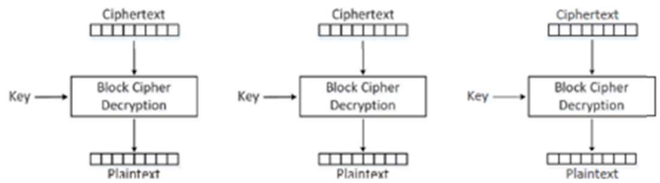
Advanced Encryption Standard (AES)

- AES is a block cipher
- 128-bit block size: plaintext 128 bits----- > ciphertext 128 bits.
- Three different key sizes: 128, 192, and 256 bits

Electronic Codebook (ECB) Mode



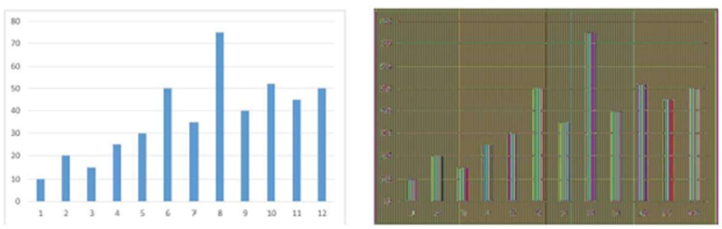
(a) Electronic Codebook (ECB) mode encryption



(b) Electronic Codebook (ECB) mode decryption

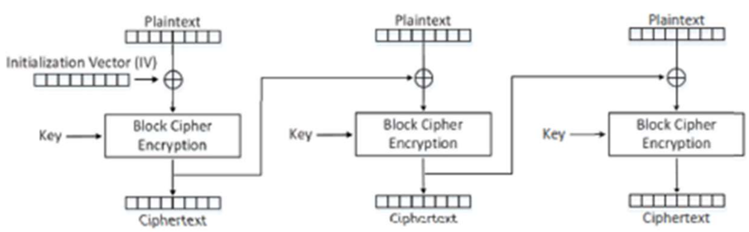
ECB is not enough for file encryption

plaintext

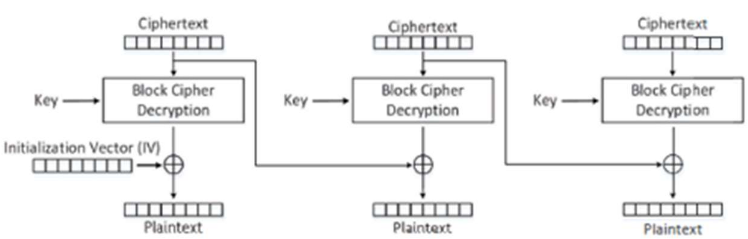


(a) The original image (pic_original.bmp) (b) The encrypted image (pic_encrypted.bmp

Cipher Block Chaining (CBC) Mode



(a) Cipher Block Chaining (CBC) mode encryption



(b) Cipher Block Chaining (CBC) mode decryption

- With different IVs, a plaintext file will encrypt to completely different ciphertext files.
- Previous blocks affect ciphertexts of subsequent blocks.
- Problem for ECB is avoid

Padding

- The encryption is block-by-block. E.g., block-size=128-bit in AES-128
- If plaintext Mis 120-bit, then it should be padded with 8 bit to make a block. This is called **plaintext padding**.
- You can not simply add 00000000 to M because receiver can not know the message is M or M0 or M00 or ….

Programming using Crypto APIs

endec_AES.py

```
#!/usr/bin/python3

from Crypto.Cipher import AES
from Crypto.Util import Padding
from Crypto.Random import get_random_bytes


key_hex_string = '00112233445566778899AABBCCDDEEFF00112233445566778899AABBCCDDEEFF'
key = bytes.fromhex(key_hex_string)
iv = get_random_bytes(16)
data = b'COMP8677 CRYPTO Lecture'
print(data.hex())

# Encrypt the entire data
cipher = AES.new(key, AES.MODE_CBC, iv)                  1
ciphertext = cipher.encrypt(Padding.pad(data, 16))       2
print("Ciphertext: {0}\n".format(ciphertext.hex()))

# Decrypt the ciphertext
cipher = AES.new(key, AES.MODE_CBC, iv)          3
plaintext = cipher.decrypt(ciphertext)           4
print("Plaintext: {0}".format(Padding.unpad(plaintext, 16)))
```

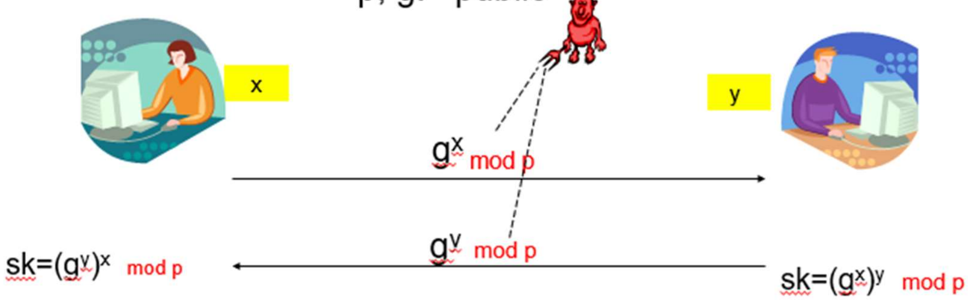
- Line:
  1. Initialize cipher
  2. Encrypt the entire data with padding, blocksize=16bytes
  3. Initialize cipher for decryption
  4. Decrypt (without unpadding)

Diffie-Hellman Key Exchange

- Alice and Bob want to share a secret key K.
- They know public parameters:
  - Big prime number p
  - A number g<p (e.g., g=2).

Diffie-Hellman key exchange

p, g: public



Discrete Logarithm Assumption:

- g^x mod p=====**hard**==== > **x**.