**ASSIGNMENT NO: A6**

**1. TITLE:**

A Write a program in python/ Java/ Scala/ C++/ HTML5 to implement password data encryption. Use encryption method overloading (any to methods studied)

**2. PREREQUISITES:**

* Latest version of 64 Bit Operating Systems Open Source Fedora-19.
* Python 2.7

**3. OBJECTIVE:**

* To implement password data encryption.
* To study and Use encryption method overloading**.**

**4. Mathematical Model:**

Let S be the solution perspective of the class Weather Report such that

S={s, e, i, o, f, DD, NDD, success, failure}

s=Start of program

e = the end of program

i=Plaintext data password

o= Encrypted data password

f= Found/Not Found

DD=Deterministic data

NDD=NULL

Success- plaintext encrypted

Failure- plaintext not encrypted

Computational Model

**5. Theory:**

**“Encryption”** is the conversion of electronic data into another form, called cipher text, which cannot be easily understood by anyone except authorized parties.

The primary purpose of encryption is to protect the confidentiality of digital data stored on computer systems or transmitted via the Internet or other computer networks. Modern encryption algorithms play a vital role in the security assurance of IT systems and communications as they can provide not only confidentiality, but also the following key elements of security:

* + Authentication: the origin of a message can be verified.
  + Integrity: proof that the contents of a message have not been changed since it was sent.
  + Non-repudiation: the sender of a message cannot deny sending the message.
  + While security is an afterthought for many PC users, it’s a major priority for businesses of any size. It has to be when the Phenomenon Institute tells us that security breaches are costing companies millions every year.
  + Even if you don’t have millions to lose, protecting what you do have should be a high priority.
  + There are several forms of security technology available, but encryption is one that everyday computer users should know about.
  + How Encryption Works
  + Encryption is an interesting piece of technology that works by scrambling data so it is unreadable by unintended parties. Let’s take a look at how it works with the email-friendly software PGP (or GPG for you open source people).

**Password Hashing**

hash("hello") = 2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824

hash("hbllo") = 58756879c05c68dfac9866712fad6a93f8146f337a69afe7dd238f3364946366

hash("waltz") = c0e81794384491161f1777c232bc6bd9ec38f616560b120fda8e90f383853542

Hash algorithms are one way functions. They turn any amount of data into a fixed-length "fingerprint" that cannot be reversed. They also have the property that if the input changes by even a tiny bit, the resulting hash is completely different (see the example above). This is great for protecting passwords, because we want to store passwords in a form that protects them even if the password file itself is compromised, but at the same time, we need to be able to verify that a user's password is correct.

The general workflow for account registration and authentication in a hash-based account system is as follows:

1. The user creates an account.
2. Their password is hashed and stored in the database. At no point is the plain-text (unencrypted) password ever written to the hard drive.
3. When the user attempts to login, the hash of the password they entered is checked against the hash of their real password (retrieved from the database).
4. If the hashes match, the user is granted access. If not, the user is told they entered invalid login credentials.
5. Steps 3 and 4 repeat everytime someone tries to login to their account.

In step 4, never tell the user if it was the username or password they got wrong. Always display a generic message like "Invalid username or password." This prevents attackers from enumerating valid usernames without knowing their passwords.

It should be noted that the hash functions used to protect passwords are not the same as the hash functions you may have seen in a data structures course. The hash functions used to implement data structures such as hash tables are designed to be fast, not secure. Only **cryptographic hash functions** may be used to implement password hashing. Hash functions like SHA256, SHA512, RipeMD, and WHIRLPOOL are cryptographic hash functions.

It is easy to think that all you have to do is run the password through a cryptographic hash function and your users' passwords will be secure. This is far from the truth. There are many ways to recover passwords from plain hashes very quickly. There are several easy-to-implement techniques that make these "attacks" much less effective. To motivate the need for these techniques, consider this very website. On the front page, you can submit a list of hashes to be cracked, and receive results in less than a second. Clearly, simply hashing the password does not meet our needs for security.

**Adding Salt**

Lookup tables and rainbow tables are ways to break hashes, only work because each password is hashed the exact same way. If two users have the same password, they'll have the same password hashes. We can prevent these attacks by randomizing each hash, so that when the same password is hashed twice, the hashes are not the same.

We can randomize the hashes by appending or prepending a random string, called a salt, to the password before hashing. As shown in the example above, this makes the same password hash into a completely different string every time. To check if a password is correct, we need the salt, so it is usually stored in the user account database along with the hash, or as part of the hash string itself.

The salt does not need to be secret. Just by randomizing the hashes, lookup tables, reverse lookup tables, and rainbow tables become ineffective. An attacker won't know in advance what the salt will be, so they can't pre-compute a lookup table or rainbow table. If each user's password is hashed with a different salt, the reverse lookup table attack won't work either.

To Store a Password

1.Generate a long random salt using a CSPRNG.

2.generate hash using standard cryptographic hash function such as SHA256

3.key stretching using PBKDF2 inpust name = standard cryptographic hash function such as SHA256, password = hash generated, salt = salt, round = 10000 aleast, dklen = 64 atleast length of derived key

4.Save both the salt and the hash in the user's database record.

To Validate a Password

1.Retrieve the user's salt and hash from the database.

2.Prepend the salt to the given password and hash it using the same hash function.

3.Compare the hash of the given password with the hash from the database. If they match, the password is correct. Otherwise, the password is incorrect.

**Making Password Cracking Harder: Slow Hash Functions**

Salt ensures that attackers can't use specialized attacks like lookup tables and rainbow tables to crack large collections of hashes quickly, but it doesn't prevent them from running dictionary or brute-force attacks on each hash individually. High-end graphics cards (GPUs) and custom hardware can compute billions of hashes per second, so these attacks are still very effective. To make these attacks less effective, we can use a technique known as **key stretching**.

The idea is to make the hash function very slow, so that even with a fast GPU or custom hardware, dictionary and brute-force attacks are too slow to be worthwhile. The goal is to make the hash function slow enough to impede attacks, but still fast enough to not cause a noticeable delay for the user.

Key stretching is implemented using a special type of CPU-intensive hash function. Don't try to invent your own–simply iteratively hashing the hash of the password isn't enough as it can be parallelized in hardware and executed as fast as a normal hash. Use a standard algorithm like [PBKDF2](http://en.wikipedia.org/wiki/PBKDF2) or[bcrypt](http://en.wikipedia.org/wiki/Bcrypt).

These algorithms take a security factor or iteration count as an argument. This value determines how slow the hash function will be. For desktop software or smartphone apps, the best way to choose this parameter is to run a short benchmark on the device to find the value that makes the hash take about half a second. This way, your program can be as secure as possible without affecting the user experience.

If you use a key stretching hash in a web application, be aware that you will need extra computational resources to process large volumes of authentication requests, and that key stretching may make it easier to run a Denial of Service (DoS) attack on your website. I still recommend using key stretching, but with a lower iteration count. You should calculate the iteration count based on your computational resources and the expected maximum authentication request rate. The denial of service threat can be eliminated by making the user solve a CAPTCHA every time they log in. Always design your system so that the iteration count can be increased or decreased in the future.

If you are worried about the computational burden, but still want to use key stretching in a web application, consider running the key stretching algorithm in the user's browser with JavaScript. The [Stanford JavaScript Crypto Library](http://crypto.stanford.edu/sjcl/) includes PBKDF2. The iteration count should be set low enough that the system is usable with slower clients like mobile devices, and the system should fall back to server-side computation if the user's browser doesn't support JavaScript. Client-side key stretching does not remove the need for server-side hashing. You must hash the hash generated by the client the same way you would hash a normal password.

**RSA**

RSA is a public-key encryption algorithm and the standard for encrypting data sent over the internet. It also happens to be one of the methods used in our PGP and GPG programs.

Unlike Triple DES, RSA is considered an asymmetric algorithm due to its use of a pair of keys. You’ve got your public key, which is what we use to encrypt our message, and a private key to decrypt it. The result of RSA encryption is a huge batch of mumbo jumbo that takes attackers quite a bit of time and processing power to break.

**Algorithm**

The RSA algorithm involves four steps: [key](https://en.wikipedia.org/wiki/Key_(cryptography)) generation, key distribution, encryption and decryption.

RSA involves a *public key* and a [*private key*](https://en.wikipedia.org/wiki/Private_key)*.* The public key can be known by everyone and is used for encrypting messages. The intention is that messages encrypted with the public key can only be decrypted in a reasonable amount of time using the private key.

The basic principle behind RSA is the observation that it is practical to find three very large positive integers *e*,*d* and *n* such that with [modular exponentiation](https://en.wikipedia.org/wiki/Modular_exponentiation) for all *m*:

(m^e)^d \equiv m \pmod{n}

and that even knowing *e* and *n* or even *m* it can be extremely difficult to find *d*.

Additionally, for some operations it is convenient that the order of the two exponentiations can be changed and that this relation also implies:

(m^d)^e \equiv m \pmod{n}

**Key distribution**[[edit](https://en.wikipedia.org/w/index.php?title=RSA_(cryptosystem)&action=edit&section=4)]

To enable [Bob](https://en.wikipedia.org/wiki/Alice_and_Bob) to send his encrypted messages, [Alice](https://en.wikipedia.org/wiki/Alice_and_Bob) transmits her public key (*n*, *e*) to Bob via a reliable, but not necessarily secret route. The private key is never distributed.

**Encryption**[[edit](https://en.wikipedia.org/w/index.php?title=RSA_(cryptosystem)&action=edit&section=5)]

Suppose that Bob would like to send message *M* to Alice.

He first turns *M* into an integer *m*, such that 0 ≤ *m* < *n* and gcd(*m*, *n*) = 1 by using an agreed-upon reversible protocol known as a [padding scheme](https://en.wikipedia.org/wiki/RSA_(cryptosystem)#Padding_schemes). He then computes the ciphertext *c*, using Alice's public key *e*, corresponding to

 c \equiv m^e \pmod{n} 

This can be done efficiently, even for 500-bit numbers, using modular exponentiation. Bob then transmits *c* to Alice.

**Decryption**[[edit](https://en.wikipedia.org/w/index.php?title=RSA_(cryptosystem)&action=edit&section=6)]

Alice can recover *m* from *c* by using her private key exponent *d* by computing

c^d \equiv (m^e)^d \equiv m \pmod{n}

Given *m*, she can recover the original message *M* by reversing the padding scheme.

**Key generation**[[edit](https://en.wikipedia.org/w/index.php?title=RSA_(cryptosystem)&action=edit&section=7)]

The keys for the RSA algorithm are generated the following way:

1. Choose two distinct [prime numbers](https://en.wikipedia.org/wiki/Prime_number) *p* and *q*.
   * For security purposes, the integers *p* and *q* should be chosen at random, and should be similar in magnitude but 'differ in length by a few digits'[[2]](https://en.wikipedia.org/wiki/RSA_(cryptosystem)#cite_note-rsa-2) to make factoring harder. Prime integers can be efficiently found using a [primality test](https://en.wikipedia.org/wiki/Primality_test).
2. Compute *n* = *pq*.
   * *n* is used as the [modulus](https://en.wikipedia.org/wiki/Modular_arithmetic) for both the public and private keys. Its length, usually expressed in bits, is the [key length](https://en.wikipedia.org/wiki/Key_length).
3. Compute φ(*n*) = φ(*p*)φ(*q*) = (*p* − 1)(*q* − 1) = *n* − (*p* + *q* − 1), where φ is [Euler's totient function](https://en.wikipedia.org/wiki/Euler%27s_totient_function). This value is kept private.
4. Choose an integer *e* such that 1 < *e* < φ(*n*) and [gcd](https://en.wikipedia.org/wiki/Greatest_common_divisor" \o "Greatest common divisor)(*e*, φ(*n*)) = 1; i.e., *e* and φ(*n*) are [coprime](https://en.wikipedia.org/wiki/Coprime).
5. Determine *d* as *d* ≡ *e*−1 (mod φ(*n*)); i.e., *d* is the [modular multiplicative inverse](https://en.wikipedia.org/wiki/Modular_multiplicative_inverse) of *e* (modulo φ(*n*))

* This is more clearly stated as: solve for *d* given *d*⋅*e* ≡ 1 (mod φ(*n*))
* *e* having a short [bit-length](https://en.wikipedia.org/wiki/Bit-length) and small [Hamming weight](https://en.wikipedia.org/wiki/Hamming_weight) results in more efficient encryption – most commonly 216 + 1 = 65,537. However, much smaller values of *e* (such as 3) have been shown to be less secure in some settings.[[13]](https://en.wikipedia.org/wiki/RSA_(cryptosystem)#cite_note-Boneh-13)
* *e* is released as the public key exponent.
* *d* is kept as the private key exponent.

The *public key* consists of the modulus *n* and the public (or encryption) exponent *e*. The *private key* consists of the modulus *n* and the private (or decryption) exponent *d*, which must be kept secret. *p*, *q*, and φ(*n*) must also be kept secret because they can be used to calculate *d*.

* An alternative, used by [PKCS#1](https://en.wikipedia.org/wiki/PKCS1), is to choose *d* matching *de* ≡ 1 (mod λ) with λ = lcm(*p* − 1, *q* − 1), where lcm is the [least common multiple](https://en.wikipedia.org/wiki/Least_common_multiple). Using λ instead of φ(*n*) allows more choices for *d*. λ can also be defined using the [Carmichael function](https://en.wikipedia.org/wiki/Carmichael_function), λ(*n*).

Since any common factors of (p − 1) and (q − 1) are present in the factorisation of pq − 1,[[14]](https://en.wikipedia.org/wiki/RSA_(cryptosystem)#cite_note-14) it is recommended that (p − 1) and (q − 1) have only very small common factors, if any besides the necessary 2.[[15]](https://en.wikipedia.org/wiki/RSA_(cryptosystem)#cite_note-15)[[2]](https://en.wikipedia.org/wiki/RSA_(cryptosystem)#cite_note-rsa-2)[[16]](https://en.wikipedia.org/wiki/RSA_(cryptosystem)#cite_note-16)

Note: The authors of RSA carry out the key generation by choosing d and then computing e as the [modular multiplicative inverse](https://en.wikipedia.org/wiki/Modular_multiplicative_inverse) of d (modulo φ(n)). Since it is beneficial to use a small value for e (i.e. 65,537) in order to speed up the encryption function, current implementations of RSA, such as [PKCS#1](https://en.wikipedia.org/wiki/PKCS1) choose e and compute d instead.[[2]](https://en.wikipedia.org/wiki/RSA_(cryptosystem)#cite_note-rsa-2)[[17]](https://en.wikipedia.org/wiki/RSA_(cryptosystem)#cite_note-17)

**Example**[[edit](https://en.wikipedia.org/w/index.php?title=RSA_(cryptosystem)&action=edit&section=8)]

Here is an example of RSA encryption and decryption. The parameters used here are artificially small, but one can also [use OpenSSL to generate and examine a real keypair](https://en.wikibooks.org/wiki/Cryptography/Generate_a_keypair_using_OpenSSL).

1. Choose two distinct prime numbers, such as

p = 61 and q = 53

1. Compute *n* = *pq* giving

n = 61\times 53 = 3233

1. Compute the [totient](https://en.wikipedia.org/wiki/Totient) of the product as φ(*n*) = (*p* − 1)(*q* − 1) giving

\varphi(3233) = (61 - 1)(53 - 1) = 3120

1. Choose any number 1 < *e* < 3120 that is [coprime](https://en.wikipedia.org/wiki/Coprime) to 3120. Choosing a prime number for *e* leaves us only to check that *e* is not a divisor of 3120.

Let e = 17

1. Compute *d*, the [modular multiplicative inverse](https://en.wikipedia.org/wiki/Modular_multiplicative_inverse) of *e* (mod φ(*n*)) yielding,

d = 2753

Worked example for the modular multiplicative inverse:

d \times e \; \operatorname{mod}\; \varphi(n) = 1

2753 \times 17  \; \operatorname{mod}\; 3120 = 1

The **public key** is (*n* = 3233, *e* = 17). For a padded [plaintext](https://en.wikipedia.org/wiki/Plaintext) message *m*, the encryption function is

c(m) = m^{17} \; \operatorname{mod}\; 3233

The **private key** is (*d* = 2753). For an encrypted [ciphertext](https://en.wikipedia.org/wiki/Ciphertext" \o "Ciphertext) *c*, the decryption function is

m(c) = c^{2753} \; \operatorname{mod}\; 3233

For instance, in order to encrypt *m* = 65, we calculate

c = 65^{17} \; \operatorname{mod}\; 3233 = 2790 

To decrypt *c* = 2790, we calculate

m = 2790^{2753} \; \operatorname{mod}\; 3233 = 65 

Both of these calculations can be computed efficiently using the [square-and-multiply algorithm](https://en.wikipedia.org/wiki/Square-and-multiply_algorithm) for [modular exponentiation](https://en.wikipedia.org/wiki/Modular_exponentiation). In real-life situations the primes selected would be much larger; in our example it would be trivial to factor *n*, 3233 (obtained from the freely available public key) back to the primes *p* and *q*. Given *e*, also from the public key, we could then compute *d* and so acquire the private key.

Practical implementations use the [Chinese remainder theorem](https://en.wikipedia.org/wiki/Chinese_remainder_theorem) to speed up the calculation using modulus of factors (mod *pq* using mod *p* and mod *q*).

The values *dp*, *dq* and *q*inv, which are part of the private key are computed as follows:

\begin{align}
           d_p &= d\; \operatorname{mod}\; (p-1) = 2753 \; \operatorname{mod}\; (61-1) = 53 \\
           d_q &= d\; \operatorname{mod}\;(q-1) = 2753 \; \operatorname{mod}\; (53-1) = 49 \\
  q_\text{inv} &= q^{-1} \; \operatorname{mod}\; p = 53^{-1} \; \operatorname{mod}\; 61 = 38 \\
               &\Rightarrow (q_\text{inv} \times q) \; \operatorname{mod}\; p = 38 \times 53 \; \operatorname{mod}\; 61= 1
\end{align}

Here is how *dp*, *dq* and *q*inv are used for efficient decryption. (Encryption is efficient by choice of a suitable *d* and *e* pair)

\begin{align}
  m_1 &= c^{d_p} \; \operatorname{mod}\; p = 2790^{53} \; \operatorname{mod}\; 61 = 4 \\
  m_2 &= c^{d_q} \; \operatorname{mod}\; q = 2790^{49} \; \operatorname{mod}\; 53 = 12 \\
    h &= (q_\text{inv} \times (m_1 - m_2)) \; \operatorname{mod}\; p = (38 \times -8) \; \operatorname{mod}\; 61 = 1 \\
    m &= m_2 + h \times q = 12 + 1 \times 53 = 65
\end{align}

**Conclusion:**

Hence we have designed to design a program to implement password data encryption using encryption method overloading.

File: encryptPass.py

import hashlib

import sys

import os

import binascii

try:

   hash\_name = sys.argv[1]

except IndexError:

   print('1st arg\n'+ str(hashlib.algorithms\_guaranteed)+'\n2nd arg username','\n3rd arg password')

else:

   try:

       username = sys.argv[2]

       passwd = sys.argv[3]

       password = bytes(passwd,'UTF-8')

   except IndexError:

       print('using default password: password')

       password = b'password'

   salt = os.urandom(32)

   chef\_salt = binascii.hexlify(salt)

   filename = username+"\_salt.key"

   salT\_file = open(username+"\_salt.key","w")

   print("Salt\n"+chef\_salt.decode('utf-8'))

   salT\_file.write(chef\_salt.decode('utf-8'))

   salT\_file.close()

   dk = hashlib.pbkdf2\_hmac(hash\_name, password, chef\_salt, 100000,128)

   store\_hash = binascii.hexlify(dk)

   filename = username+"\_hash.hash"

   hash\_file = open(username+"\_hash.hash","w")

   print("hash\n"+store\_hash.decode('utf-8'))

   hash\_file.write(store\_hash.decode('utf-8'))

   hash\_file.close()

File: verifyPass.py

import hashlib

import sys

import os

import binascii

try:

   hash\_name = sys.argv[1]

except IndexError:

   print('Specify the hash name as the first argument.')

else:

   try:

       username = sys.argv[2]

       passwd = sys.argv[3]

       password = bytes(passwd,'UTF-8')

   except IndexError:

       print('using default password: password')

       password = b'password'

   #password = bytes(passwd,'UTF-8')

   salT\_file = open(username+"\_salt.key")

   chef\_salt = salT\_file.read()

   print("Salt\n"+chef\_salt)

   dk = hashlib.pbkdf2\_hmac(hash\_name, password, bytes(chef\_salt,'UTF-8'), 100000,128)

   hashk = binascii.hexlify(dk)

   new\_hash = hashk.decode('utf-8')

   hash\_file = open(username+"\_hash.hash")

   old\_hash  = hash\_file.read()

   print("hash\n"+old\_hash)

   if new\_hash == old\_hash:

    print("The user "+username+" has used valid password")

   else:

    print("Incorrect username or password")

'''

envy@envy-vm:~/be-2$ python3 encryptPass.py

1st arg

{'sha1', 'md5', 'sha384', 'sha224', 'sha512', 'sha256'}

2nd arg username

3rd arg password

envy@envy-vm:~/be-2$ python3 encryptPass.py sha256 cybersecurity 1l0v3cy83453cu41ty

Salt

becf85585f32a32fa21e2c600ec1b90ee91e71407cd47c20fc661afa96d7837e

hash

9040251af5493315fdecb02661e44a321cebba914fa1e540d507e748f2f36525f918471baa12c9f48c62d8f7ba2c13a94e76c3a7e06819f236104e91618058b330135466106e28ccdb67c95d494d87edbe252768d4e9885ff012dcde01fab9ae0c04e1f14380eb3e0ac54e6dd8387e2bcc291b5560eff3b8b8074a8d3957f9df

envy@envy-vm:~/be-2$

envy@envy-vm:~/be-2$

envy@envy-vm:~/be-2$ python3 verifyPass.py sha256 cybersecurity 1l0v3cy83453cu41ty

Salt

becf85585f32a32fa21e2c600ec1b90ee91e71407cd47c20fc661afa96d7837e

hash

9040251af5493315fdecb02661e44a321cebba914fa1e540d507e748f2f36525f918471baa12c9f48c62d8f7ba2c13a94e76c3a7e06819f236104e91618058b330135466106e28ccdb67c95d494d87edbe252768d4e9885ff012dcde01fab9ae0c04e1f14380eb3e0ac54e6dd8387e2bcc291b5560eff3b8b8074a8d3957f9df

The user cybersecurity has used valid password

envy@envy-vm:~/be-2$ python3 verifyPass.py sha256 cybersecurity l0v3cy83453cu41ty

Salt

becf85585f32a32fa21e2c600ec1b90ee91e71407cd47c20fc661afa96d7837e

hash

9040251af5493315fdecb02661e44a321cebba914fa1e540d507e748f2f36525f918471baa12c9f48c62d8f7ba2c13a94e76c3a7e06819f236104e91618058b330135466106e28ccdb67c95d494d87edbe252768d4e9885ff012dcde01fab9ae0c04e1f14380eb3e0ac54e6dd8387e2bcc291b5560eff3b8b8074a8d3957f9df

Incorrect username or password

envy@envy-vm:~/be-2$

'''