

Identificación de algoritmos de hash

Para identificar a que algoritmo de hasheo corresponde cada archivo se utilizaron dos páginas web: <https://www.tunnelsup.com/hash-analyzer/> y https://hashes.com/en/tools/hash_identifier . Para esto se verificó en dichas páginas ingresando una contraseña aleatoria de cada archivo, lo que permitió validar los algoritmos de hasheo.

- Archivo 1: MD5

Hash:	f3e611a0b7112ee8bd053aa023ec678d
Salt:	Not Found
Hash type:	MD5 or MD4
Bit length:	128
Character length:	32
Character type:	hexidecimal

- Archivo 2: MD5 plus salt

Hash:	da0084f67f4a09a33b405ce70c036a85
Salt:	vs1wEzivhlOPYe/1
Hash type:	MD5 or MD4 : plus salt
Bit length:	128
Character length:	32
Character type:	hexidecimal

- Archivo 3: MD5 plus salt

Hash:	de5e3da259b60fa6c136802eaac34b87
Salt:	h!s8Q#g31MB0J7PM
Hash type:	MD5 or MD4 : plus salt
Bit length:	128
Character length:	32
Character type:	hexidecimal

- Archivo 4: NTLM

✓ Possible identifications: [Decrypt Hashes](#)

b2ec285deb6809af10c90aa40a420ee3 – Possible algorithms: NTLM

- - Archivo 5: sha512crypt



Obtención de textos planos

Para obtener los textos planos de los archivos con contraseñas hashadas se utilizó la función `crack_files`:

```
def crack_files(files):
    for file in files:
        command = "hashcat -m " + str(file['mode']) + ' -o ' +
file['output'] + ' ' + file['path'] + ' ' + DICCIONARIO_2 + ' --force'
        start_time = time.time()
        os.system(command)
        print("---- %s seconds ----" % (time.time() - start_time))
```

Dicha función recibe de entrada un arreglo de archivos a crackear, donde cada archivo es un diccionario que posee la ruta donde se encuentra ubicado el archivo (`path`), el código del algoritmo de hashado para identificar en hashcat `mode` y el archivo de salida a utilizar (`output`). Para poder descifrar las contraseñas se utiliza hashcat.

Este algoritmo fue utilizado bajo una máquina virtual con las siguientes características:

- Sistema operativo: Ubuntu 18.04LTS
- Memoria: 4096 MB

Mientras que la máquina base posee las siguientes características:

- Sistema operativo: MacOS Catalina 10.15.6
- Procesador: 3,6 GHz Quad-Core Intel Core i3
- Memoria: 16 GB 2400 MHz DDR4
- Gráfica: AMD Radeon R9 280X 3 GB

Resultados

Al ejecutar la función `crack_files()` se obtuvieron los siguientes tiempos para crackear cada uno de los archivos y la cantidad total de hash que posee cada archivo y cuantos de estos se lograron crackear:

- Archivo 1:
 - Tiempo: 2,35 segundos
 - Hash totales: 1000

- Hash crackeados: 1000

```

Session.....: hashcat
Status.....: Cracked
Hash.Type.....: MD5
Hash.Target.....: ./archivos/Hashes/archivo_1
Time.Started.....: Thu Nov 12 00:42:46 2020 (0 secs)
Time.Estimated...: Thu Nov 12 00:42:46 2020 (0 secs)
Guess.Base.....: File (./archivos/diccionarios/diccionario_2.dict)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 913.6 kH/s (0.26ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered.....: 1000/1000 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.....: 316096/316096 (100.00%)
Rejected.....: 0/316096 (0.00%)
Restore.Point....: 315392/316096 (99.78%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1....: zrt60096 -> zzzzzzzzzz

Started: Thu Nov 12 00:42:44 2020
Stopped: Thu Nov 12 00:42:47 2020
--- 2.353501081466675 seconds ---

```

- Archivo 2:
 - Tiempo: 3,34 segundos
 - Hash totales: 1000
 - Hash crackeados: 1000

```

Session.....: hashcat
Status.....: Cracked
Hash.Type.....: md5($pass.$salt)
Hash.Target.....: ./archivos/Hashes/archivo_2
Time.Started.....: Thu Nov 12 00:42:48 2020 (0 secs)
Time.Estimated...: Thu Nov 12 00:42:48 2020 (0 secs)
Guess.Base.....: File (./archivos/diccionarios/diccionario_2.dict)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 933.0 kH/s (0.29ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered.....: 1000/1000 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.....: 316096/316096 (100.00%)
Rejected.....: 0/316096 (0.00%)
Restore.Point....: 315392/316096 (99.78%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1....: zrt60096 -> zzzzzzzzzz

Started: Thu Nov 12 00:42:47 2020
Stopped: Thu Nov 12 00:42:50 2020
--- 3.3440675735473633 seconds ---

```

- Archivo 3:
 - Tiempo: 45.51 segundos
 - Hash totales: 1000

- Hash crackeados: 1000

```

Session.....: hashcat
Status.....: Cracked
Hash.Type.....: md5($pass.$salt)
Hash.Target.....: ./archivos/Hashes/archivo_3
Time.Started.....: Thu Nov 12 00:42:51 2020 (44 secs)
Time.Estimated...: Thu Nov 12 00:43:35 2020 (0 secs)
Guess.Base.....: File (./archivos/diccionarios/diccionario_2.dict)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 1934.1 kH/s (0.87ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered.....: 1000/1000 (100.00%) Digests, 1000/1000 (100.00%) Salts
Progress.....: 316094592/316096000 (100.00%)
Rejected.....: 0/316094592 (0.00%)
Restore.Point....: 315392/316096 (99.78%)
Restore.Sub.#1...: Salt:997 Amplifier:0-1 Iteration:0-1
Candidates.#1....: zrt60096 -> zzzzzzzzzz

Started: Thu Nov 12 00:42:50 2020
Stopped: Thu Nov 12 00:43:35 2020
--- 45.51594591140747 seconds ---

```

- Archivo 4:
 - Tiempo: 2,34 segundos
 - Hash totales: 1000
 - Hash crackeados: 1000

```

Session.....: hashcat
Status.....: Cracked
Hash.Type.....: NTLM
Hash.Target.....: ./archivos/Hashes/archivo_4
Time.Started.....: Thu Nov 12 02:38:32 2020 (1 sec)
Time.Estimated...: Thu Nov 12 02:38:33 2020 (0 secs)
Guess.Base.....: File (./archivos/diccionarios/diccionario_2.dict)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 989.7 kH/s (0.16ms) @ Accel:1024 Loops:1 Thr:1 Vec:8
Recovered.....: 1000/1000 (100.00%) Digests, 1/1 (100.00%) Salts
Progress.....: 316096/316096 (100.00%)
Rejected.....: 0/316096 (0.00%)
Restore.Point....: 315392/316096 (99.78%)
Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1
Candidates.#1....: zrt60096 -> zzzzzzzzzz

Started: Thu Nov 12 02:38:31 2020
Stopped: Thu Nov 12 02:38:33 2020
--- 2.3429672718048096 seconds ---

```

- Archivo 5:
 - Tiempo: 6647.11 segundos
 - Hash totales: 20

- Hash crackeados:20

```
Session.....: hashcat
Status.....: Cracked
Hash.Type.....: sha512crypt $6$, SHA512 (Unix)
Hash.Target.....: ./archivos/Hashes/archivo_5
Time.Started.....: Thu Nov 12 00:43:47 2020 (1 hour, 50 mins)
Time.Estimated...: Thu Nov 12 02:34:26 2020 (0 secs)
Guess.Base.....: File (./archivos/diccionarios/diccionario_2.dict)
Guess.Queue.....: 1/1 (100.00%)
Speed.#1.....: 413 H/s (7.14ms) @ Accel:256 Loops:64 Thr:1 Vec:4
Recovered.....: 20/20 (100.00%) Digests, 20/20 (100.00%) Salts
Progress.....: 5824768/6321920 (92.14%)
Rejected.....: 0/5824768 (0.00%)
Restore.Point....: 291072/316096 (92.08%)
Restore.Sub.#1...: Salt:12 Amplifier:0-1 Iteration:4992-5000
Candidates.#1....: Tryder -> TTARKAS

Started: Thu Nov 12 00:43:41 2020
Stopped: Thu Nov 12 02:34:28 2020
--- 6647.111760139465 seconds ---
```

Diferencias

Mediante estas pruebas se puede apreciar que el algoritmo md5 es el algoritmo más rápido en descifrar las contraseñas, esto se debe a que es un algoritmo que no es resistente a colisiones y la capacidad de computo de la máquina es capaz de poder descifrar estas contraseñas rápidamente. En los hashes del archivo 2 a pesar de poseer un salt el algoritmo md5, este resultado descifrarse en un tiempo casi idéntico al md5 sin salt, debido a que al conocer el salt de las contraseñas solo resta saber la porción del hash que no corresponde al salt, al contrario de lo que ocurre en el algoritmo md5 plus salt del archivo 3, ya que cada contraseña en ese archivo posee un salt distinto.

El algoritmo más seguro se utilizó en el archivo 5, que utiliza un algoritmo **sha512crypt**, esto se evidencia en el tiempo que tomó descifrar tan solo 20 contraseñas debido a que este tipo de hash emplea más mecanismos para aumentar el tiempo requerido en descifrar los mensajes.

Re-cifrado

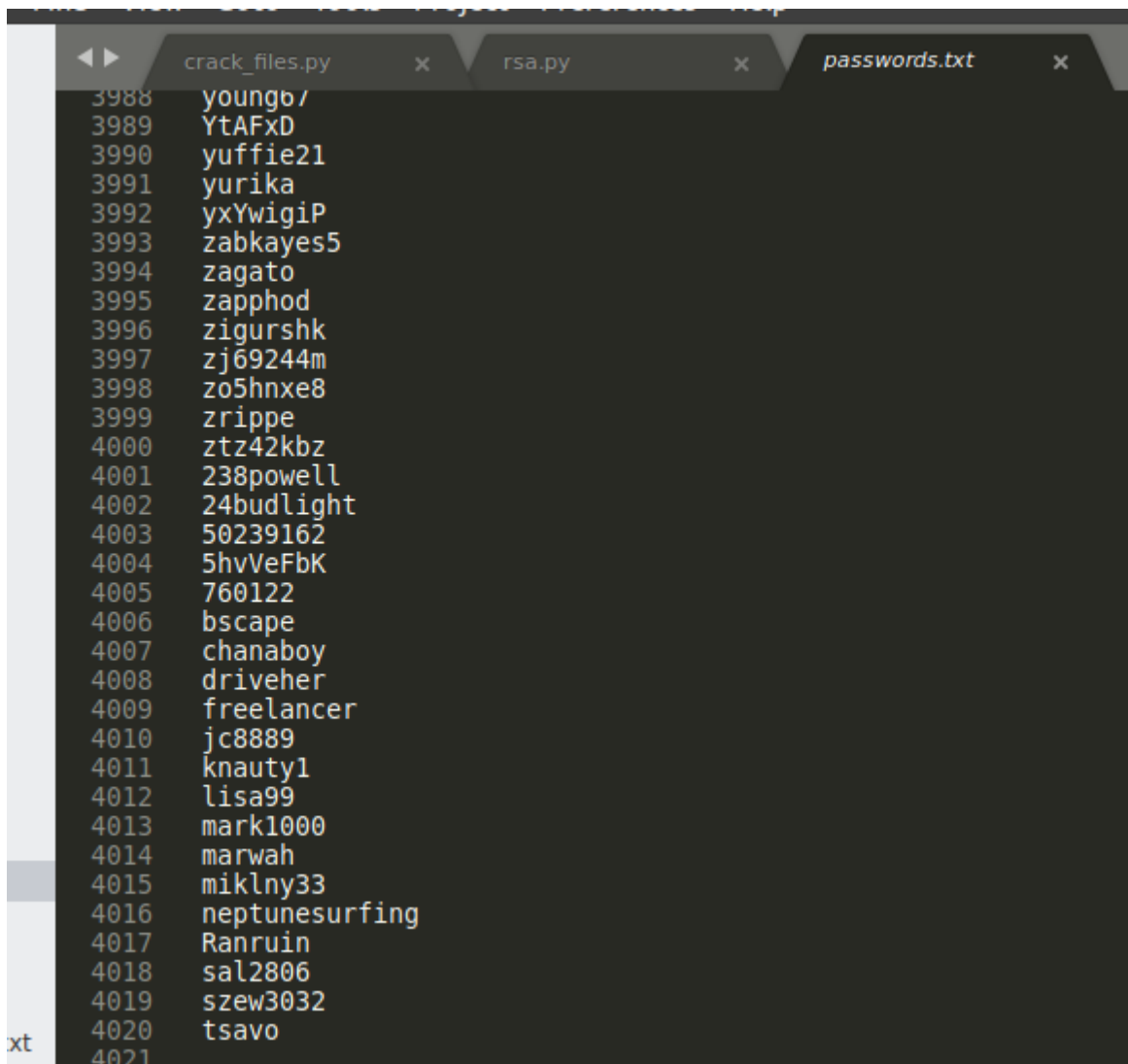
Para volver a cifrar las contraseñas se utilizará el algoritmo **Bcrypt** por las siguientes razones:

- Salt: El que este algoritmo haga uso de **salt** permite siempre generar distintos cifrados para la misma contraseña.
- Lentitud: Bcrypt fue ideado para ser un algoritmo lento, lo que dificulta los ataques por fuerza bruta, al contrario de lo que podría ser un algoritmo de la familia SHA-2, que son diseñados para ser rápidos facilitando así estos ataques.

En primera instancia se utilizó la función **create_file()** para generar un archivo único con las 4020 contraseñas en texto plano:

```
def create_file(files):
    new_file = open('passwords.txt', 'a')
    for file in files:
        password_file = open(file, 'r')
        for line in password_file:
```

```
clean_password = line.split(':')[1]
new_file.write(clean_password)
```

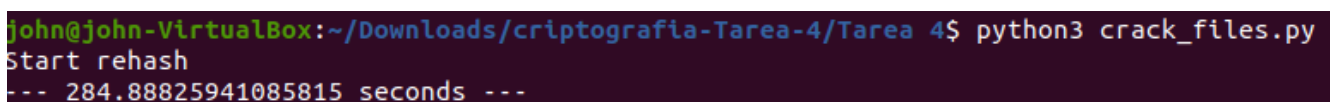


```
crack_files.py x rsa.py x passwords.txt x
3988 youngb/
3989 YtAFxD
3990 yuffie21
3991 yurika
3992 yxYwigiP
3993 zabkayes5
3994 zagato
3995 zapphod
3996 zigurshk
3997 zj69244m
3998 zo5hnxe8
3999 zrippe
4000 ztz42kbz
4001 238powell
4002 24budlight
4003 50239162
4004 5hvVeFbK
4005 760122
4006 bscape
4007 chanaboy
4008 driveher
4009 freelancer
4010 jc8889
4011 knautyl
4012 lisa99
4013 mark1000
4014 marwah
4015 miklny33
4016 neptunesurfing
4017 Ranruin
4018 sal2806
4019 szew3032
4020 tsavo
4021
```

Luego, para realizar el re-cifrado en bcrypt se programó la función `rehash()` :

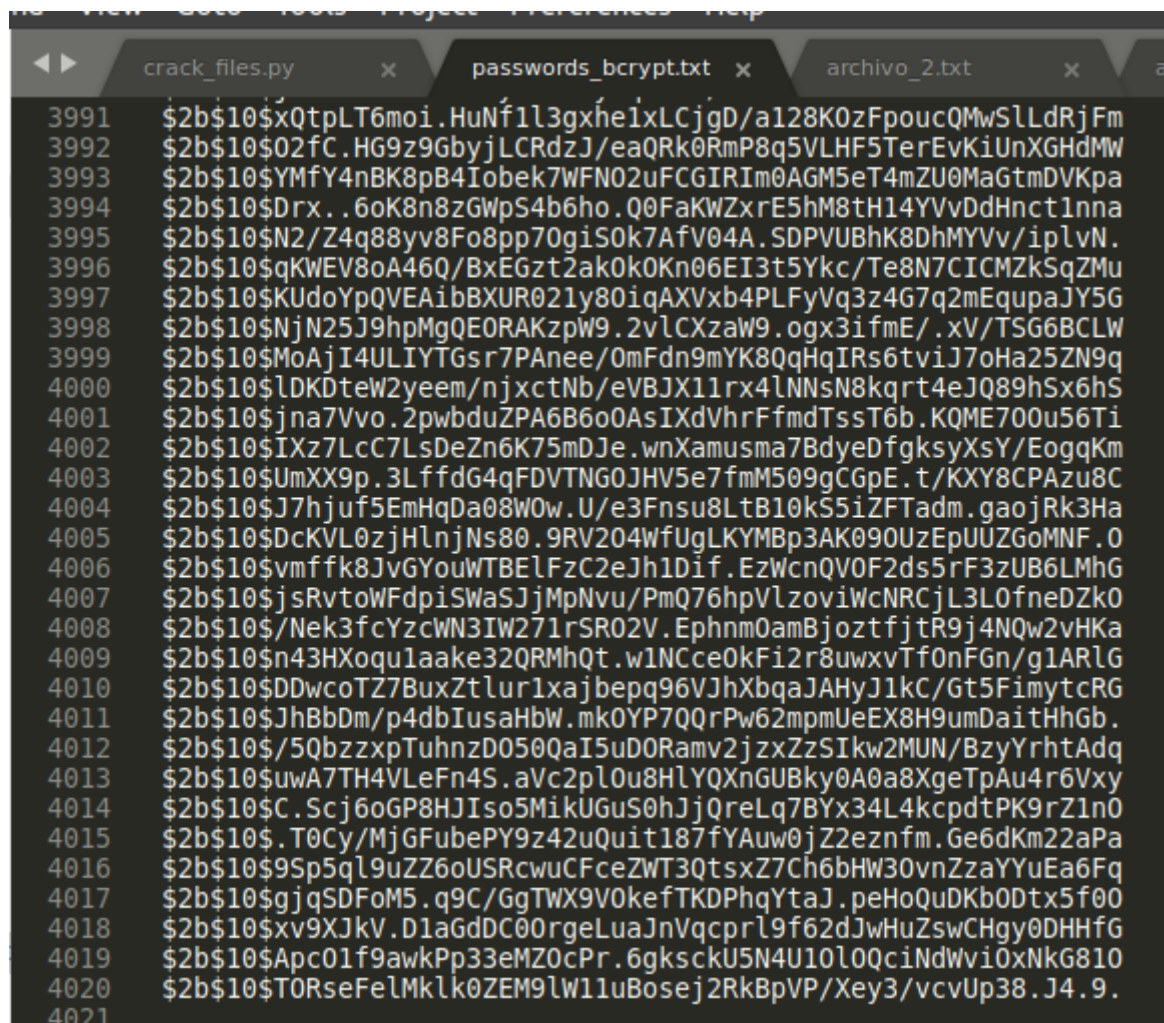
```
def rehash():
    rehashed_passwords = open('passwords_bcrypt.txt', 'a')
    passwords = open('passwords.txt', 'r')
    print('Start rehash')
    start_time = time.time()
    for password in passwords:
        hashed = bcrypt.hashpw(password = password.strip().encode('utf-8'), salt = bcrypt.gensalt(rounds=10))
        rehashed_passwords.write(hashed.decode('utf-8') + '\n')
    print("--- %s seconds ---" % (time.time() - start_time))
```

La que, tras ser ejecutada, llevó un tiempo de 284.88 segundos en completar la operación.



```
john@john-VirtualBox:~/Downloads/criptografia-Tarea-4/Tarea 4$ python3 crack_files.py
Start rehash
--- 284.88825941085815 seconds ---
```


Estas contraseñas se encuentran en el archivo `passwords_bcrypt.txt` el cual contiene las 4020 contraseñas encriptadas utilizando el algoritmo `Bcrypt`.



```
crack_files.py x passwords_bcrypt.txt x archivo_2.txt x ar
3991 $2b$10$X0tpLT6moi.HuNf1l3gxhe1xLCjgD/a128K0zFpoucQMwSLdRjFm
3992 $2b$10$02fC.HG9z9GbyjLCRdzJ/eaQRk0RmP8q5VLHF5TerEvKiUnXGHdMW
3993 $2b$10$YMfY4nBK8pB4Iobek7WfN02uFCGIRIm0AGM5eT4mZU0MaGtmDVKpa
3994 $2b$10$Drx..6oK8n8zGwP54b6ho.Q0FaKWZxrE5hM8tH14YVvDdHnctlnna
3995 $2b$10$N2/Z4q88yv8Fo8pp70giS0k7AfV04A.SDPVUBhK8DhMYVv/iplvN.
3996 $2b$10$qKEV8oA46Q/BxEGzt2ak0k0Kn06EI3t5Ykc/Te8N7CICMzkSqZMu
3997 $2b$10$KUdoYpQVEAibBXUR021y80iqAXVxb4PLFyVq3z4G7q2mEqupaJY5G
3998 $2b$10$Njn25J9hpMgQEORAKzpw9.2vLCXzaW9.ogx3ifmE/.xV/TS66BCLW
3999 $2b$10$MoAjI4ULIYTgsr7PAnee/OmFdn9mYK8QqHqIRs6tviJ7oHa25ZN9q
4000 $2b$10$LDKdteW2yeem/njxctNb/eVBJX11rx4lNNsN8kgqt4eJQ89hSx6hS
4001 $2b$10$jna7Vvo.2pwbduZPA6B6o0AsIXdVhrFmdTssT6b.KQME700u56Ti
4002 $2b$10$IXz7LcC7LsDeZn6K75mDJe.wnXamusma7BdyeDfgksyXsY/EogqKm
4003 $2b$10$UmXX9p.3Lffdg4qFDVTNG0JHV5e7fmM509gCGpE.t/KXY8CPAzu8C
4004 $2b$10$J7hjuf5EmHqDa08W0w.U/e3Fnsu8LtB10kS5iZFTadm.gaojRk3Ha
4005 $2b$10$DcKVL0zjHlnjNs80.9RV204WfUgLKymbp3AK090UzEpUUZGoMNF.0
4006 $2b$10$vmffk8JvGYouWTBElFzC2eJh1Dif.EzWcnQV0F2ds5rF3zUB6LMhG
4007 $2b$10$jsRvt0WFdpISWaSJjMpNvu/PmQ76hpVlzoviWcNRCjL3L0fneDZk0
4008 $2b$10$/Nek3fcYzcWN3IW271rSR02V.Ephnm0amBjoztfjtr9j4NQw2vHKA
4009 $2b$10$43HXoqu1aake32QRMhQt.w1NCce0kFi2r8uwxvTf0nFGn/g1ARlG
4010 $2b$10$DDwcoT7BuxZtlur1xajbepq96VJhXbqaJAHyJ1kC/Gt5FimytCRG
4011 $2b$10$JhBbDm/p4dbIusaHbw.mk0YP7QqrPw62mpmUeEX8H9umDaitHhGb.
4012 $2b$10$/5QbzzxpTuhnzD050QaI5uD0Ramv2jzxZzSIkw2MUN/BzyYrhtAdq
4013 $2b$10$uwA7TH4VLeFn4S.aVc2pl0u8HLYQXnGUBky0A0a8XgeTpAu4r6Vxy
4014 $2b$10$.Scj6oGP8HJIso5MikUGuS0hJjQreLq7BYx34L4kcpdtPK9rZln0
4015 $2b$10$.T0Cy/MjGFubePY9z42uQuit187fYAuw0jZ2eznfm.Ge6dKm22aPa
4016 $2b$10$9Sp5ql9uZZ6oUSRcwCFceZWT30tsxZ7Ch6bHW30vnZzaYYuEa6Fq
4017 $2b$10$gjqSDFoM5.q9C/GgTWX9V0kefTKDPhqYtaJ.peHoQuDKb0Dtx5f00
4018 $2b$10$xv9XJkV.D1aGdDC00rgeLuaJnVqcprl9f62dJwHuZswCHgy0DHHfG
4019 $2b$10$Apc01f9awkPp33eMZ0cPr.6gksckU5N4U10l0QciNdWvi0xNkG810
4020 $2b$10$T0RseFelMklk0ZEM9lW1luBosej2RkBPVP/Xey3/vcvUp38.J4.9.
4021
```

Cifrado asimétrico

Se implementó el algoritmo de cifrado asimétrico RSA.

El proceso para generar la llave pública y privada se compone de diversos pasos:

1. Seleccionar dos números primos `p` y `q`. Para tener fuertes llaves públicas y privadas se recomienda utilizar números primos muy largos.
2. Computar `n`, que es el resultado de multiplicar `p` con `q`. $n = p * q$
3. Computar $\lambda(n)$ que corresponde a la función de Charmichael. $\lambda(n) = (p - 1) * (q - 1)$
4. Computar la llave de encriptación (`e`), que corresponde a un número tal que el máximo común divisor entre `e` y $\lambda(n)$ sea 1, es decir, que `e` y $\lambda(n)$ sean coprimos.
5. Computar la llave de descifrado (`d`) que corresponde al inverso multiplicativo modular de `e` y se obtiene utilizando el algoritmo extendido de Euclides ($d \equiv e^{-1} \pmod{\lambda(n)}$).

Esto se programó en la función `generate_keys()`

```
def generate_keys(p, q):
    if not (is_prime(p) and is_prime(q)):
        raise ValueError('Ambos numeros deben ser primos')
    elif p == q:
```

```

        raise ValueError('Ambos numeros deben ser distintos')

# Computa n
n = p * q

# Computa phi
phi = (p - 1) * (q - 1)

# Calcula la llave de encriptacion
e = random.randrange(1, phi) # Escoge un numero entero
g = greatest_common_divisor(e, phi) # Calcula el maximo comun divisor
while g != 1: # Calcula numero hasta encontrar coprimos
    e = random.randrange(1, phi)
    g = greatest_common_divisor(e, phi)

# Calcula la llave de desencriptacion
d = modular_multiplicative_inverse(e, phi)[1]
d = d % phi
if d < 0:
    d += phi

# Llave publica (e), llave privada(d) y n
return (e, d, n)

```

Solicitud de llaves

Para el proceso de solicitud de llaves se utilizó una conexión por sockets entre dos archivos, para esto el script `crack_files.py` actuará como cliente, conectándose vía socket al script `rsa.py` que actua como servidor.

Cliente:

```

# Crea el TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Conecta al servidor
server_address = ('localhost', 10009)
print('connecting to {} port {}'.format(*server_address))
sock.connect(server_address)

BUFFER_SIZE = 1024

try:
    # Envia la peticion de llave publica
    message = b'REQUEST_PUBLIC_KEY'
    print('Sending {}'.format(message))
    sock.sendall(message)

    # Recibe la respuesta
    public_key = recv_message(sock)
    n = ''
    if public_key:

```



```

        public_key, separator, n = public_key.partition(b':')
        print('public_key {}'.format(public_key.decode('utf-8')))
        passwords_bcrypt = open('passwords_bcrypt.txt', 'r')
        encrypt_passwords(public_key, n, passwords_bcrypt)
        send_file(sock)

```

```

finally:
    print('closing socket')
    sock.close()

```

Servidor:

```

# Crea el socket TCP/IPa
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Enlaza el socket al puerto
server_address = ('localhost', 10009)
print('Server started on {} port {}'.format(*server_address))
sock.bind(server_address)

# Espera por una conexion
sock.listen(1)

BUFFER_SIZE = 1024

while True:
    print('Waiting for a connection')
    connection, client_address = sock.accept()

    try:
        print('connection from', client_address)
        p,q = get_prime(), get_prime()
        while p == q:
            q = get_prime()
        # Recibe la peticion y envia las llaves
        while True:
            request = connection.recv(BUFFER_SIZE)
            print('request: ', request)
            if request.decode('utf-8') == 'REQUEST_PUBLIC_KEY':
                public_key, private_key, n = generate_keys(p, q)
                print ('Public key: ', public_key)
                print ('Private key: ', private_key)
                print ('n: ', n)
                public_key = str(public_key) + ':' + str(n)
                package = str(sys.getsizeof(public_key)) + ':' +
str(public_key)
                connection.sendall(bytes(package.encode('utf-8')))
                filename = receive_file(connection)
                decrypt_file(str(private_key), n, filename)
                save_to_sqlite()
            else:
                print('no data required')

```

```

        break

finally:
    # Cierra la coneccion
    connection.close()

```

Cifrado de hashes utilizando RSA

Se utilizó la función en el archivo cliente que permite cifrar los hashes en Bcrypt pero ahora aplicándoles RSA.

```

def encrypt_passwords(public_key, n, passwords):
    rsa_passwords = open('rsa_passwords.txt', 'a')
    print('=== Encriptando ===')
    for password in passwords:
        # Convierte cada letra en la contraseña a numeros, basado en el
        # caracter utilizando a^b mod(m)
        cipher_password = [pow(ord(char), int(public_key.decode('utf-8'))),
                           int(n.decode('utf-8')) for char in password]
        cipher_password = ''.join(map(lambda x: str(x), cipher_password))
        rsa_passwords.write(cipher_password + '\n')
    print('=== Finalizo encriptado ===')

```

Este algoritmo opera con la llave pública y el `n` que se obtienen vía socket del archivo servidor, donde una vez cifrados mediante RSA se envía el archivo con las 4020 contraseñas hashadas al servidor mediante la función :

```

def send_file(sock):
    filename = 'rsa_passwords.txt'
    # Obtiene el tamaño del archivo en bytes
    filesize = os.path.getsize(filename)
    print('=== Enviando archivo ===')
    # Envía el tamaño y nombre del archivo
    sock.send(f'{filesize}:{filename}'.encode())
    progress = tqdm.tqdm(range(filesize), f'Sending {filename}', unit="B",
                          unit_scale=True, unit_divisor=1024)
    with open(filename, 'rb') as file:
        for _ in progress:
            # Lee los bytes desde el archivo
            bytes_read = file.read(4096)
            if not bytes_read:
                # Termina la transmisión
                break
            # Envía los bytes
            sock.sendall(bytes_read)
            # Actualiza la barra de progreso
            progress.update(len(bytes_read))
    print('=== Finaliza envío de archivo ===')

```

Una vez enviado este archivo se finaliza el proceso en el cliente.

En el servidor se cuenta con la función `receive_file()` que es la encargada de procesar el recibimiento del archivo con contraseñas desde el cliente para ser guardadas en el archivo `ser-rsa_passwords.txt` y posteriormente ser descriptadas, utilizando la llave privada y el `n`, mediante la función `decrypt_file()`.

```
def receive_file(connection):
    print('=== Recibiendo arvchivo ===')
    received = connection.recv(4096).decode()
    filesize, filename = received.split(':')
    # Elimina la path absoluta (siesque esta)
    filename = os.path.basename(filename)

    # Convierte el tamaño del archivo a entero
    filesize = int(filesize)

    # Actualiza el nombre del archivo para diferenciarlo del archivo
    # enviado por el cliente
    filename = 'server-' + filename

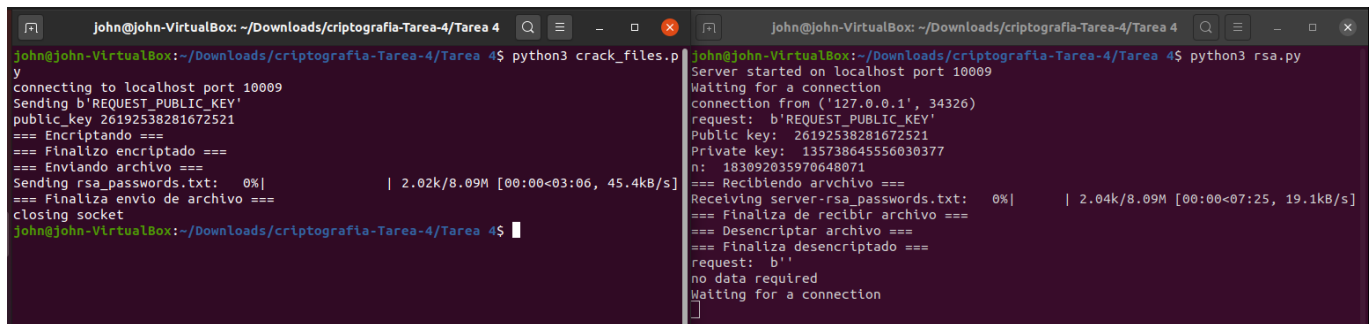
    # Comienza a recibir el archivo
    progress = tqdm.tqdm(range(filesize), f'Receiving {filename}',
unit='B', unit_scale=True, unit_divisor=1024 )
    with open(filename, 'wb') as file:
        for _ in progress:
            # Lee 1024 bytes desde el socket
            bytes_read = connection.recv(4096)
            if not bytes_read:
                # Nada es recibido
                break
            # Escribe al archivo los bytes recibidos
            file.write(bytes_read)
            # Actualiza la barra de progreso
            progress.update(len(bytes_read))
    print('=== Finaliza de recibir archivo ===')
    return filename
```

```
def decrypt_file(private_key, n, filename):
    print('=== Desencriptar archivo ===')
    file = open(filename, 'r')
    for rsa_password in file:
        bdecrypt_password = decrypt(private_key, n, rsa_password)
    print('=== Finaliza desencriptado ===')

def decrypt(private_key, n, cipher_text):
    try:
        password = [chr(pow(ord(char), private_key, n)) for char in
cipher_text]
        return ''.join(password)
    except TypeError as e:
```

pass

El procedimiento completo en simultaneo mediante sockets se puede evidenciar en la siguiente captura:



```
john@john-VirtualBox: ~/Downloads/criptografia-Tarea-4/Tarea 4$ python3 crack_files.py
y
Connecting to localhost port 10009
Sending b'REQUEST_PUBLIC_KEY'
public key 26192538281672521
=== Encriptando ===
=== Finalizo encriptado ===
=== Enviando archivo ===
Sending rsa_passwords.txt: 0% | 2.02k/8.09M [00:00<03:06, 45.4kB/s]
=== Finaliza envio de archivo ===
closing socket
john@john-VirtualBox:~/Downloads/criptografia-Tarea-4/Tarea 4$

john@john-VirtualBox:~/Downloads/criptografia-Tarea-4/Tarea 4$ python3 rsa.py
Server started on localhost port 10009
Waiting for a connection
connection from ('127.0.0.1', 34326)
request: b'REQUEST_PUBLIC_KEY'
Public key: 26192538281672521
Private key: 135738645556030377
n: 183092035970648071
=== Recibiendo archivo ===
Receiving server-rsa_passwords.txt: 0% | 2.04k/8.09M [00:00<07:25, 19.1kB/s]
=== Finaliza de recibir archivo ===
=== Descriptar archivo ===
=== Finaliza descriptado ===
request: b''
no data required
Waiting for a connection
```

Almacenamiento en SQLite

Finalmente, las contraseñas en encriptación Bcrypt se almacenan desde el servidor en una base de datos SQLite. Función que no se implementó.

Enlaces

Github: <https://github.com/JohnBidwellB/criptografia/tree/Tarea-4/Tarea%204>