udp Escuela de Informática
y Telecomunicaciones
FACULTAD DE INGENIERÍA Y CIENCIAS

21 de junio, 2021

Criptografía y seguridad en redes

Tarea 4: Cifrado asimétrico

Integrante : Joaquín Lagos Martínez

Profesor : Nicolás Boettcher
Ayudante : Macarena Velásquez

1 Características

1.1 Software

- Python versión 3.9.5
- Librería Python: PyCryptodomex versión 3.10.1
- Librería Python: Libnum 1.7.1
- pip versión 21.1.2
- Hashcat 6.2.2
- Algoritmo asimétrico: Okamoto-Uchiyama

1.2 Hardware

- Procesador: Intel Core i5-6200U @ 2.3 GHz
- Memoria RAM: 8 GB DDR3 1600 MHz
- Almacenamiento: SSD SATA3
- iGPU: Intel HD Graphics 510
- dGPU: NVIDIA GeForce 940M 2GB

2 Okamoto-Uchiyama:

Es un algoritmo asimétrico creado por Tatsuaki Okamoto and Shigenori Uchiyama en 1988.

3 Identificación de *hash*

Se utilizaron diferentes métodos de inmortificación. En el primer archivo se utilizó la herramienta de la página TunnelsUP, donde se ingresó uno de los *hash*, arrojando como resultado una posible respuesta **MD5** O **MD4**. Para identificar si coincide con alguna de las dos, se intentó crackear con **hashcat**, primero con **md5**, el cual funcionó, por lo tanto corresponde a ese algoritmo.

Para el segundo y tercer archivo se utilizó el mismo método. Arrojando en ambos **MD5** o **MD4** utilizando *salt*, con la diferencia que el del segundo archivo tiene *salt* fijo y el del tercero es variable. **Hashcat** no tiene la opción de crackear **MD4** + *salt*, por lo tanto se prueba con **MD5** + *salt* de la misma forma que en el caso anterior, resultando exitosamente.

Para el cuarto archivo, se utilizo la página anterior, debido a que TunnelsUP arrojó nuevamente **MD5** o **MD4**, lo cual fue probado mas tarde y no fue posible realizar el crackeo. El algoritmo arrojado por Hashes.com, otra herramienta para identificar algoritmos de resumen, fue NTLM, soportado por Hashcat, por lo que se intentó crackear y resultó en éxito.

Para el quinto se utilizó la página Hashes.com, ya que **TunnelsUP** no arrojó un resultado. Esta página arrojó como posible algoritmo *SHA512crypt*, que es, a grandes rasgos, un **SHA512** con un *salt* y el ID del *hash* utilizado (6).

4 Crackeo de los archivos

4.1 Archivo 1:

El siguiente código fue ejecutado en la consola de Windows, en el directorio de la carpeta contenedora de Hashcat.

```
.\hashcat.exe -m 0 -a 0 -D 2
--outfile-format=2 -o ..\op1.txt
..\Hashes\archivo_1
..\diccionarios\diccionario_1.dict ..\diccionarios\diccionario_2.dict
```

Parámetros utilizados:

- -m 0 : corresponde al algoritmo *hash* a crackear. En este archivo corresponde a **MD5** que tiene el ID '0'.
- -a 0 : establece el modo de ataque. El '0' corresponde al ataque por diccionario o *'straight'*.
- -D 2: asigna el dispositivo a utilizar. '2' corresponde a la tarjeta gráfica dedicada.
- - -output-format = 2 : determina el formato del archivo de salida, siendo '2' el texto plano de la contraseña.
- -o ..\op1.txt : ruta del archivo de salida.
- ..\Hashes\archivo_1 : ruta del archivo con los *hash*.
- ..\diccionarios\diccionario_1.dict : ruta al diccionario 1.
- ..\diccionarios\diccionario_2.dict : ruta al diccionario 2.

Para los demás archivos, se utiliza el mismo modo de ataque, dispositivo, formato de salida y los diccionarios.

4.1.1 Archivo 2

```
.\hashcat.exe -m 10 -a 0 -D 2
--outfile-format=2 -o ..\op2.txt
..\Hashes\archivo_2
..\diccionarios\diccionario_1.dict ..\diccionarios\diccionario_2.dict
```

Parámetros utilizados:

- -m 10 : corresponde al algoritmo *hash* a crackear. En este archivo corresponde a **MD5 + salt** que tiene el ID '10'.
- -o ..\op2.txt : ruta del archivo de salida.
- ..\Hashes\archivo_2 : ruta del el archivo con los *hash*.

4.1.2 Archivo 3

```
.\hashcat.exe -m 10 -a 0 -D 2
--outfile-format=2 -o ..\op3.txt
..\Hashes\archivo_3
..\diccionarios\diccionario_1.dict ..\diccionarios\diccionario_2.dict
```

Parámetros utilizados:

- -m 10 : corresponde al algoritmo *hash* a crackear. En este archivo corresponde a **MD5 + salt** que tiene el ID '10'.
- -o ..\op3.txt : ruta del archivo de salida.
- ..\Hashes\archivo_3 : ruta del el archivo con los *hash*.

4.1.3 Archivo 4

```
.\hashcat.exe -m 1000 -a 0 -D 2
--outfile-format=2 -o ..\op4.txt
..\Hashes\archivo_4
..\diccionarios\diccionario_1.dict ..\diccionarios\diccionario_2.dict
```

Parámetros utilizados:

- -m 1000 : corresponde al algoritmo *hash* a crackear. En este archivo corresponde a **NTLM** que tiene el ID '1000'.
- -o ..\op4.txt : ruta del archivo de salida.
- ..\Hashes\archivo_4 : ruta del archivo con los *hash*.

4.1.4 Archivo 5

```
.\hashcat.exe -m 1800 -a 0 -D 2
--outfile-format=2 -o ..\op5.txt
..\Hashes\archivo_5
..\diccionarios\diccionario_1.dict ..\diccionarios\diccionario_2.dict
```

Parámetros utilizados:

- -m 1000 : corresponde al algoritmo *hash* a crackear. En este archivo corresponde a **sha512crypt** que tiene el ID '1800'.
- -o ..\op5.txt : ruta del archivo de salida.
- ..\Hashes\archivo_5 : ruta del archivo con los *hash*.

4.1.5 Resultados:

Los resultados al obtener las contraseñas de cada archivo, ya implementado en el programa de Python, son los siguientes:

- Archivo 1 (MD5): 8.6 segundos.
- Archivo 2 (MD5 + SALT): 8.7 segundos.
- Archivo 3 (MD5 + SALT variable): 15.7 segundos.
- Archivo 4 (NTLM): 9.3 segundos.
- Archivo 5 (sha512crypt): 877.5 segundos.

El tiempo de los dos primeros archivos son similares, lo cual lleva a pensar que usando un *salt* fijo no es muy efectivo para retrasar un ataque por diccionario. Esto se puede deber a el poco control que se tiene de la asignación de recursos del computador y puede haber asignado mayor prioridad a otro proceso mientras se ejecutaba este. El mayor tiempo en el archivo 3 se debe a el *salt*, que ya no es fijo y no existe un patrón que pueda utilizar Hashcat para tardar menos. Con respecto al cuarto archivo, también tiene un tiempo similar ya que usa **NTLM** es muy similar a **MD4**. El quinto algoritmo es el que más tardó, por la cantidad de rondas de **SHA-512** utilizadas.

5 Códigos en Python

Al ejecutar el código *cliente.py*, este empezará a crackear los archivos de texto, el proceso tardará cerca de 15 minutos. Antes de que este termine debe ser ejecutado el servidor para que se establezca la conexión.

5.1 Servidor

Este se encarga de generar las llaves tanto públicas como privadas, enviar la pública al cliente, recibir los nuevos *hash* cifrados, los descifra y los guarda en un archivo de texto. Además, crea el canal de conexión para comunicarse con el cliente a través de *sockets*. A continuación se explicará el código paso a paso.

Las librería utilizadas son *socket*, para la conexión; *time*, para medir los tiempos de ejecución; *pickle*, convierte objetos en Python en un *stream* de bytes para enviarlos por *sockets*; *binascii*, para convertir el *hash* hexadecimal a bytes y viceversa. Además, se importa el archivo que contiene el algoritmo de cifrado asimétrico, localizado en el mismo directorio que ambos códigos Python.

```
1  import socket
2  from Okamoto_Uchiyama import *
3  import time
4  import pickle
5  import binascii
```

Se establece el número de bits de los números primos utilizados para generar las llaves y luego se generan con *'gen_key(prime)'*, que toma como argumento el largo definido. Se generan tres llaves públicas ($\boldsymbol{n}$, $\boldsymbol{g}$ y $\boldsymbol{h}$) y dos privadas ($\boldsymbol{p}$ y $\boldsymbol{q}$).

```
8  # Largo de los numeros primos usados para generar las laves
9  prime = 1200
```

```
16 # Se generan las llaves publicas y privadas
17 n,g,h,p,q=gen_key(prime)
```

Se inicializa un objeto *socket* y se le asocia la dirección 'localhost' y el puerto 8000. Se deja escuchando a la conexión del cliente.

```
30 # Se inicializa socket
31 sckt = socket.socket()
32 # Se asocia la direccion y puerto
33 sckt.bind(('localhost',8000))
34 # Permite la entrada de un cliente a la vez
35 sckt.listen(1)
```

Se abre un ciclo *While* infinito, el cual contendrá todo el proceso restante. Se toma el tiempo inicial, se conecta el cliente. Luego se mandan las tres llaves públicas convertidas desde números enteros a *strings* y luego codificadas en bytes. Se crea un arreglo que guardará los *hash* encriptados del cliente. Se inicializa la variable 'data' como un *string* vacío.

```
37  while True:
38      T_incial = time.time()
39
40      # Acepta al cliente
41      conexion, adress = sckt.accept()
42      print("nueva conexion establecida",'\n')
43      print(conexion)
44
45      # Envía las tres llaves publicas
46      conexion.send(str(n).encode('ascii'))
47      time.sleep(1)
48      conexion.send(str(g).encode('ascii'))
49      time.sleep(1)
50      conexion.send(str(h).encode('ascii'))
51
52      # Crea el arreglo donde se guardará lo recibido por cliente
53      ciphers_received = []
54      data = ''
```

Se abre un ciclo *While* que recibirá bloques de máximo 2024 bytes. Se extrae el largo del mensaje del bloque recibido, que corresponde a los últimos 10 bytes, y se compara con el largo actual (menos los 10 ya extraídos). Se verifica si es el mensaje de término, si lo es se sale del ciclo, si no, agrega el mensaje cifrado a un arreglo. Luego de recibir toda la información y finaliza el *while*. Se crea un arreglo donde se agregarán los *hash*.

```
56      while True:
57          # Se recibe mensaje cifrado
58          data = conexion.recv(2042)
59          # Se extrae el largo del mensaje recibido
60          data_len = int(data[:10])
61          # Se verifica que el largo de el mensaje coincida con el valor enviado en el encabezado
62          if len(data) - 10 == data_len:
63              # Se verifica si es el mensaje de término, si lo es, se sale del loop for
64              if str(pickle.loads(data[10:])) == 'end':
65                  break
66              # Se agrega el mensaje cifrado a el arreglo, sin el encabezado
67              ciphers_received.append(str(pickle.loads(data[10:])))
68          else:
69              print('Error al recibir: '+ str(pickle.loads(data[10:])))
70      print('Se recibió toda la información!','\n')
71      data_array = []
```

Se recorre cada posición del arreglo con los *ciphers*, cada uno correspondiendo a un *hash* cifrado. El '*cipher*' se transforma a entero y se utiliza la función *decrypt*, con parámetros el mensaje cifrado, la llave pública $\boldsymbol{g}$ y un llave privada. Se decodifican los bytes del

output del desencriptado a formato hexadecimal y se ignoran los bytes que no se pueden decodificar, se conservan los 128 caracteres de largo del *hash* y finalmente se agrega cada uno en forma de *string* al arreglo *data_array*.

```
74      # Se recorre el arreglo
75      for line in range(len(ciphers_received)):
76
77          cipher_str = ciphers_received[line]
78          # El cifrado se transforma a entero
79          cipher = int(cipher_str)
80          # Se descifra con una de las llaves públicas y una de las privadas
81          hash_bytes = decrypt(cipher, p, g)
82          # Se decodifican y se corta trunca a un tamaño de 128, tamaño del output del algoritmo hash
83          try:
84              hash = binascii.hexlify(hash_bytes)[:128]
85              data_array.append(str(hash))
86          except:
87              print('Error al decodificar')
```

Se recorre el arreglo, se elimina cada 'b' y comillas dejadas por la decodificación y se escribe en el archivo *data_recieved.txt*

```
90      with open('data_recieved.txt', 'w') as file:
91          for item in data_array:
92              item = item[2:130]
93              file.write('%s\n' % item)
94
```

Finalmente, se calcula el tiempo desde que se conectó el cliente y cierra la conexión y el servidor.

```
96      print('Terminó el proceso, hashes guardados en el archivo hash_client_end.txt','\n')
97      # Se termina la conexión con el cliente
98      T_final = time.time()
99      conexion.close()
100
101     T_conn = T_final - T_incial
102     print("Conexión terminada, tiempo desde inicio de la conexión: " + str(T_conn))
103     # Se cierra el servidor
104     sckt.close()
```

5.2 Cliente

El cliente deberá crackear los *hash* iniciales, usando un ataque con diccionarios proporcionados. Luego aplicará un algoritmo mas seguro que los iniciales, se conectará al servidor y recibirá las llaves públicas de generadas por este. Cifrará los nuevos *hash* y los enviará al servidor.

Se importan las librerías *socket*; *pickle*; *os*, para ejecutar comandos en la consola de Windows; *time*; el algoritmo **Okamoto-Uchiyama**; la función *hash* **SHA-512** de la librería *PyCryptodomex* y *binascii* para pasar de hexadecimal a bytes.

```
1  import socket
2  import pickle
3  import os
4  import time
5  from Okamoto_Uchiyama import *
6  from Cryptodome.Hash import SHA3_512
7  import binascii
```

Se guardan los comandos de Hashcat, descritos mas arriba, agregando al principio 'cd hashcat-6.2.2 &' para que entre a la carpeta contenedora de Hashcat.

```
11 # Comandos de Hashcat para los archivos 1-5
12 cmd1 = 'cd hashcat-6.2.2 & .\hashcat.exe -m 0 -a 0 -D 2 --outfile-format=2 -o ..\\op1.txt ..\\Hashes\\archivo_1 ..
13 cmd2 = 'cd hashcat-6.2.2 & .\hashcat.exe -m 10 -a 0 -D 2 --outfile-format=2 -o ..\\op2.txt ..\\Hashes\\archivo_2 .
14 cmd3 = 'cd hashcat-6.2.2 & .\hashcat.exe -m 10 -a 0 -D 2 --outfile-format=2 -o ..\\op3.txt ..\\Hashes\\archivo_3 .
15 cmd4 = 'cd hashcat-6.2.2 & .\hashcat.exe -m 1000 -a 0 -D 2 --outfile-format=2 -o ..\\op4.txt ..\\Hashes\\archivo_4
16 cmd5 = 'cd hashcat-6.2.2 & .\hashcat.exe -m 1800 -a 0 -D 2 --outfile-format=2 -o ..\\op5.txt ..\\Hashes\\archivo_5
```

Se ejecuta una a la vez, con la función *system* de la librería *os*. Antes y después de cada ejecución se guarda el tiempo con *time.time()*. Se calcula el tiempo demorado en cada crackeo y se imprimen.

```
18  # Se ejecuta cada comando
19  t1 = time.time()
20  os.system(cmd1)
21  t2 = time.time()
22  os.system(cmd2)
23  t3 = time.time()
24  os.system(cmd3)
25  t4 = time.time()
26  os.system(cmd4)
27  t5 = time.time()
28  os.system(cmd5)
29  t6 = time.time()
30
31  # Se calcula el tiempo demorado de cada crackeo
32  t_cmd1 = t2 - t1
33  t_cmd2 = t3 - t2
34  t_cmd3 = t4 - t3
35  t_cmd4 = t5 - t4
36  t_cmd5 = t6 - t5
```

Se inicializa socket y se conecta a la dirección y puerto del servidor, 'localhost' y 8000 respectivamente. Recibe las tres llaves publicas. Las decodifica desde 'ASCII' y se convierten a enteros.

```
49  # Se conecta al servidor
50  sckt.connect(('localhost', 8000))
51  # Recibe las tres llaves públicas
52  rec1 = sckt.recv(3000)
53  time.sleep(1)
54  rec2 = sckt.recv(3000)
55  time.sleep(1)
56  rec3 = sckt.recv(3000)
57
58  # Las decodifica a ASCII
59  dec1 = rec1.decode('ascii')
60  dec2 = rec2.decode('ascii')
61  dec3 = rec3.decode('ascii')
62
63  # Las convierte a enteros
64  pblc_key_n = int(dec1)
65  pblc_key_g = int(dec2)
66  pblc_key_h = int(dec3)
```

Se crea un arreglo con el nombre de los archivos de salida de cada uno de los crackeos.

```
73  # Arreglo con el nombre de los archivos contenedores de las contraseñas
74  files = ['op1.txt','op2.txt','op3.txt','op4.txt','op5.txt']
```

Se recorre ese arreglo con un ciclo *for*, en el cual se abre el archivo en modo lectura y se recorre cada línea. Si la línea es distinto de un *string* vacío o un $\backslash n$, el *string* de la contraseña en texto plano se codifica en 'utf-8', se le aplica **SHA3-512** y se guarda el resultado en hexadecimal. Se cifra, dándole a la función el *hash* convertido en bytes con *'unhexlify()'*, con las tres llaves públicas, dando como resultado un numero entero. Se transforma a *string* y se convierte en bytes con *pickle*. Se le agrega un encabezado de largo 10 que contiene el largo de el mensaje a enviar. Luego se manda al servidor. El tiempo calculado es el tiempo demorado en *hashear* con **SHA3-512**, que luego es impreso.

```
76  # Se recorre el arreglo files
77  for file in files:
78      # Se abre el archivo file correspondiente a en cada iteración
79      fl = open(file, 'r')
80      # Se recorre el archivo
81      tt = 0
82
83      for lines in fl:
84          # Se extrae una linea
85          line = lines.strip()
86          # Se verifica que no sea un espacio en blanco o un salto de línea
87          if line != '' and line != '\n':
88              # Se codifica cada linea a utf-8 (por defecto)
89              data = line.encode()
90              # Se aplica hash SHA3-512
91              ti = time.time_ns()
92              hash_obj = SHA3_512.new(data)
93              tf = time.time_ns()
94              tt = tt + tf - ti
95              # Se guarda el output en hexadecimal
96              hash_hex = hash_obj.hexdigest()
97              # Se cifra el hash con Okamoto-Uchiyama, utilizando las llaves públicas
98              # EL hash pasa de hexadecimal a bytes
99              cipher = encrypt(binascii.unhexlify(hash_hex), pblc_key_n, pblc_key_g, pblc_key_h)
100             # Se transforma en string
101             cipher_str = str(cipher)
102             # Se utiliza pickle para convertir el cifrado a bytes
103             data_to_send = pickle.dumps(cipher)
104             # Se le agrega un encabezado de largo 10 con el largo del mensaje a enviar
105             data_2 = bytes(f'{len(data_to_send):<{10}}', 'utf-8') + data_to_send
106             # Se envia al servidor
107             sckt.send(data_2)
108     print("Tiempo de demora en hashear del %s = %d nanosegundos" %(file ,tt))
```

Por último, se manda un mensaje de término para avisar al servidor que se terminó la transferencia y se cierra la conexión.

```
111  # Se envía un mensaje de término
112  end_msg = 'end'
113  data_to_send = pickle.dumps(end_msg)
114  data_2 = bytes(f'{len(data_to_send):<{10}}', 'utf-8') + data_to_send
115  sckt.send(data_2)
116  # Se cierra la conexión
117  sckt.close()
118
119  T_final = time.time()
120  T_conn = T_final - T_incial
121
122  print("Tiempo de conexión: " + str(T_conn) )
```

6 GitHub

Enlace del repositorio:

- https://github.com/Juax16/Tarea_4_Criptografia_2021/

7 Referencias

- https://akkadia.org/drepper/SHA-crypt.txt
- https://asecuritysite.com/encryption/ou
- https://hashes.com/en/tools/hash_identifier
- https://www.tunnelsup.com/hash-analyzer/
- https://hashcat.net/wiki/