

Trabajo práctico entregable 1

Escalera Pedro

Imán Federico

GitHub: <https://github.com/fedeiman/Seguridad-2020>

Ejercicio 1

Si considermos las palabras compuestas por los caracteres a...z,0...9 (36 caracteres si tomamos el alfabeto en inglés) y sabemos que los caracteres se pueden repetir entonces la fórmula que me indica cuantas palabras de largo N hay es 36^N . En el siguiente gráfico podemos ver que esta función se hace muy grande muy rápido y asumiendo que toma 1 ms probar cada combinación tenemos el tiempo en el eje y y el N en el eje x . Para $N = 20$ tomaría $4.23599388 \times 10^{20}$ años en probar todas las posibilidades.



Ejercicio 2

Para resolver el ejercicio 2 lo primero a conocer es el tipo de hash, buscamos información sobre un par de tools que podríamos haber usado que nos brindarían información acerca del hash con el que estamos trabajando, pero al final no fue necesario, ya que investigando en

internet, descubrimos que era un hash de tipo SHA1 de un dispositivo android, este hash para ser crackeado necesitaba de una salt la cual también encontramos en internet "f6d45822728ddb2c", investigando mas descubrimos que la password se trataba de una que contenía 8 dígitos decimales y por ende fue fácil descubrirla con la herramienta hashcat con el siguiente comando:

```
hashcat -a 3 -n 80 -u 1024 -m 5800  
941d4637d8223d958d7f2324572c7e319dcea01f:f6d45822728ddb2c ?d?d?d?d?d?d?d?d  
force
```

-a 3 significa que sera un ataque de fuerza bruta, el cual intentara con todas las combinaciones de 8 dígitos (d?d?d?d?d?d?d?d)

-n y -u son parámetros del kernel que debíamos setear con esos valores según lo encontrado en internet

-m significa el tipo de hash y 5800 es Andriod PIN para hashcat

Luego viene el hash a crackear junto con su salt.

-force es para ignorar unos warnings que aparecían.

Finalmente el PIN encontrado por hashcat luego de aproximadamente 2 hs de realizar fuerza bruta fue: 10021981

fuente: <https://blog.wirhabenstil.de/2014/08/26/breaking-samsung-android-passwordspin/>

La Hint del ejercicio por otro lado sugeria resolver el problema de una forma mas cerca a la fuerza bruta. Basicamente, el script dado como pista obtiene los modos de hashcat correspondientes para todos los hash y los deja listos para ser utilizados llamando a hashcat con el hash dado con cada uno de estos modos. Con "\$" podemos agregarle al script la funcionalidad de correr hashcat con cada uno de los modos ya que el script como esta solo hace un echo del comando que deberiamos correr (pero no lo corre). En un script de bash \$(comando) corre el comando dado en una shell distinta e introduce el output del comando donde se encontraba el \$(comando), basicamente evalua la expresion despues del signo \$.

Ejercicio 3

El proceso usado para obtener la diferencia entre las palabras del diccionario del Práctico 0 (ejercicio 5) y las listas del directorio Passwords de Seclists fue el siguiente:

Primero, procesamos nuestro diccionario de palabras con python, guardando así en una lista llamada “ekodict”

todas las palabras pero eliminando el carácter “\n” que representaba el salto de linea.

Luego definimos una función checkifappear() que toma dos valores un str y una lista. El str es el path a una carpeta contenedora de archivos que es donde deben estar todos los archivos .txt que queramos procesar (los de secList).

(Aclaración importante: el script solo funciona para los archivos dentro del directorio que le pasamos a la función como path. no para los archivos dentro de un directorio dentro del directorio del path. Esto fue así para simplificar un poco el código. El directorio passwords de secList tiene a su vez subdirectorios, por lo que se puede solucionar el problema, corriendo el código y cambiando el path o moviendo todos los archivos de los subdirectorios al directorio principal a mano. Esta ultima fue la solución usada.)

Por otro lado la lista es la lista a comprobar cuales de sus elementos no están en secList, es decir ekodict.

Finalmente la función checkifappear() lee todos los archivos .txt en el path que le pasamos elimina el carácter “\n” de salto de linea de cada palabra y las guarda en una lista “wordlist” (esto puede ser costoso en memoria)

Y por ultimo verificamos para cada palabra de “ekodict” si aparece en “wordlist”. si aparece, guardamos esta palabra en “appearWord” y finalmente para calcular la diferencia pasamos “ekodict” y “appearWord” a conjuntos y calculamos la resta de conjuntos guardandola en noAppear. Esto devuelve un conjunto con solo los elemtos de “ekodict” que no estan en “appearWord”.

Para resolver el punto b se uso la misma función cambiando el path.

Ejercicio 4

Para resolver el ej 4, primero usamos pipal que es una herramienta que devuelve los datos pedidos. Los 3 diccionarios/listas del directorio Passwords de SecLists elegidos fueron: 10-million-password-list-top-10000.txt,

10-million-password-list-top-1000.txt,

10-million-password-list-top-100.txt.

Obteniendo los siguientes resultados:

10-million-password-list-top-100.txt

Password length (count ordered)

6 = 56 (55.45%)

7 = 16 (15.84%)

8 = 15 (14.85%)

4 = 7 (6.93%)

9 = 3 (2.97%)

5 = 2 (1.98%)

10 = 2 (1.98%)

Last 4 digits (Top 10)

1111 = 3 (2.97%)

6969 = 2 (1.98%)

4321 = 2 (1.98%)

7777 = 2 (1.98%)

3456 = 1 (0.99%)

5678 = 1 (0.99%)

6789 = 1 (0.99%)

2345 = 1 (0.99%)

1234 = 1 (0.99%)

4567 = 1 (0.99%)

10-million-password-list-top-1000.txt

Password length (count ordered)

6 = 447 (44.7%)

7 = 194 (19.4%)

8 = 173 (17.3%)

4 = 84 (8.4%)

5 = 71 (7.1%)

9 = 19 (1.9%)

10 = 11 (1.1%)

11 = 1 (0.1%)

Last 4 digits (Top 10)

4321 = 9 (0.9%)

1111 = 6 (0.6%)

1234 = 5 (0.5%)

7777 = 4 (0.4%)

0000 = 4 (0.4%)

5555 = 4 (0.4%)

6969 = 3 (0.3%)

1212 = 3 (0.3%)

8888 = 3 (0.3%)

3456 = 2 (0.2%)

10-million-password-list-top-10000.txt

Password length (count ordered)

6 = 3176 (31.76%)

8 = 2972 (29.72%)

7 = 1771 (17.71%)

5 = 903 (9.03%)

4 = 810 (8.1%)

9 = 219 (2.19%)

10 = 94 (0.94%)

11 = 28 (0.28%)

12 = 21 (0.21%)

3 = 3 (0.03%)

13 = 1 (0.01%)

15 = 1 (0.01%)

16 = 1 (0.01%)

Last 4 digits (Top 10)

1987 = 193 (1.93%)

1986 = 174 (1.74%)

1988 = 158 (1.58%)

1985 = 138 (1.38%)

1990 = 123 (1.23%)

1989 = 104 (1.04%)

1991 = 73 (0.73%)

1984 = 57 (0.57%)

1983 = 55 (0.55%)

1992 = 34 (0.34%)

Luego resolvimos el ej usando una idea propia que fue programar un script en python que nos devuelva los datos pedidos.

El script lee el archivo que le pasemos como path, elimina los caracteres “\n” de salto de linea y guarda en una lista el largo de cada palabra, luego calcula cual numero fue el que mas ocurrencias tuvo en la lista, y lo devuelve, devolviendo así cual fue la longitud de palabra mas común.

Para el punto “b”, nos quedamos con los últimos 4 caracteres de cada palabra de la lista (eliminando el “\n”) y guardamos estos 4 caracteres en una lista. luego verificamos si los últimos 4 caracteres son dígitos con el método de python isdigit() y finalmente con un método de la librería collections devolvemos los 10 sufijos de 4 dígitos más usados de dichas listas

Ejercicio 5

Para este ejercicio basicamente lo que hicimos fue armar un sistema de ecuaciones con las igualdades que conocemos que se dan a la hora de crear claves RSA. Nosotros sabiamos que:

- $N_1 = P \times Q$
- $N_2 = Q \times R$
- $N_3 = P \times R$

Y despejando P podemos ver que:

- $P = \sqrt{\frac{N_3 \times N_1}{N_2}}$
- $Q = \frac{N_2}{N_3/P}$
- $R = \frac{N_3}{P}$

Con estos datos simplemente era cuestion de hacer los calculos para obtener P, Q y R. Aqui nos encontramos con algunos problemas en python ya que los N_i eran muy grandes para entrar en un float y hacer las operaciones de raiz cuadrada y division. Para solucionarlo simplemente utilizamos el hecho de que los P, Q y R son enteros y utilizamos la division de enteros de python //

y una funcion que calcula la raiz cuadrada entera de un numero dado que encontramos en internet.

Una vez que obtuvimos los datos anteriores pudimos calcular el phi de cada una de las claves y con eso el inverso modular para calcular la clave privada de cada persona. Una vez que teniamos la clave privada decodificar el mensaje fue una cuestion de elevar el CipherText

dado a la clave privada, luego convertir este decimal a hexadecimal y finalmente a ASCII para obtener el mensaje uniendo los tres mensajes:

"flag{n0_0n3_3xp3ct5_th3_sp4nish_inquisiti0n!}"

Ejercicio 6

En este ejercicio nuestra primera intencion fue la de bruteforcear la contraseña, pero despues de un tiempo nos dimos cuenta de que tardaba mucho cada conexion al servidor asi que probablemente no era al camino a seguir. Despues de correr varias veces la hint2 del ejercicio nos dimos cuenta de que el servidor tardaba mucho mas en cerrar la conexion cuando la letra enviada era la "G". Cuando medimos el tiempor pudimos ver que efectivamente con esta letra el servidor tardaba el doble (2 segundos" hasta que cerraba la conexion que con las otras letras. Esto nos llevo a creer que se trataba de un timing attack y efectivamente asi obtuvimos la siguiente letra a mano "a". Una vez que supimos que estabamos en el camino correcto armamos un script en python que iba probando cada palabra del abecedario y media cuanto tardaba la conexion con el servidor, una vez que probaba con todo el abecedario se quedaba con el caracter que mas tiempo habia tardado y lo sumaba a la contraseña encontrada hasta el momento. Como a veces habia picos en el servidor, cada vez que se encuentra un nuevo maximo se vuelve a probar con el mismo caracter para ver que la latencia no haya sido casualidad. Despues de un tiempo (2 dias que la pasamos muy mal porque no funcionaba) tambien descubrimos que era mejor correr este ataque conectado al cable de red ya que el wifi parecia agregar mucha interferencia que arruinaba los resultados. Una vez conectados por el cable de red el programa adivino la contraseña "GaAVCK9r3K" en un tiempo aceptable.

Para calcular el mejor y peor tiemp de este tipo de ataque importa el orden del abecedario y que tanta inteligencia tiene nuestro script. En particular el que nosotros entregamos tiene un tiempo constante, ya que no corta hasta que probó todo el abecedario, esto lo hicimos asi ya que no podíamos asegurar que la interferencia fuese lo suficientemente chica como para shortcircuitear el programa, es decir, cortarlo cuando encuentre la siguiente letra. Si uno pudiese elmininar la mayoria de las interferencias y despues de varias corridas a mano pudimos ver que cada letra correcta parecia sumar un segundo a lo que tardaba el servidor en cortar la conexion, por lo tanto el mejor caso posible es si el abecedario de prueba esta ordenado con la password exacta al principio y tardaria $2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10 + 11 = 65$ segundos. En el peor caso posible, donde tiene que probar todas las posibilidades, y esta al final queda $1 * 61 + 2 * 62 + 3 * 62 + 4 * 62 + 5 * 62 + 6 * 62 + 7 * 62 + 8 * 62 + 9 * 62 + 10 * 62 + 11 = 3420$ segundos, es decir 57 minutos. Es importante notar que ambos de estos resultados son teoricos y no tienen en cuenta la interferencia en la red.

Ejercicio 7

El archivo hashes.txt tiene 16.532.955 y nosotros haciendo pruebas crackeamos 6.042.717 de las siguientes maneras:

Primero con hashcat -m 0 hashes.txt rockyou-50.txt

Session...: hashcat

Status...: Exhausted

Hash.Name...: MD5

Hash.Target...: hashes.txt

Time.Started...: Tue Sep 8 20:12:35 2020 (2 secs)

Time.Estimated...: Tue Sep 8 20:12:37 2020 (0 secs)

Guess.Base...: File (rockyou-50.txt)

Guess.Queue...: 1/1 (100.00%)

Speed.#1...: 6695 H/s (3.24ms) @

Accel:1024 Loops:1 Thr:1 Vec:8

Recovered...: 8603/16532955 (0.05%) Digestss

Remaining...: 16524352 (99.95%) Digests

Recovered/Time...: CUR:N/A,N/A,N/A AVG:321898,

19313882,463533161 (Min,Hour,Day)

Progress...: 9437/9437 (100.00%)

Rejected...: 0/9437 (0.00%)

Restore.Point...: 9437/9437 (100.00%)

Restore.Sub.#1...: Salt:0 Amplifier:0-1

Iteration:0-1

Candidates.#1...: silviu -> august17

Usando el diccionario rockyou-50.txt crakeando así

8.603 hashes

Luego durante 1hs aproximadamente, utilizamos fuerza bruta. De esta manera

hashcat -m 0 -a3 hashes.txt

obteniendo como resultado

Status...: Running

Hash.Name...: MD5

Hash.Target...: hashes.txt

Time.Started...: Wed Sep 9 15:23:08 2020 (43

mins, 19 secs)

Time.Estimated...: Wed Sep 9 23:39:46 2020 (7 hours, 33 mins)

Guess.Mask...: ?1?2?2?2?2?2?2 [7]

Guess.Charset...: -1 ?l?d?u, -2 ?l?d, -3 ?l?d*!\$@_, -4 Undefined

Guess.Queue...: 7/15 (46.67%)

Speed.#1...: 4506.6 kH/s (60.53ms) @ Accel:256 Loops:1024 Thr:1 Vec:8

Recovered...: 5321783/16532955 (32.19%) Digests

Remaining...: 11211172 (67.81%) Digests

Recovered/Time...: CUR:7778,N/A,N/A AVG:9802,588146,14115510 (Min,Hour,Day)

Progress...: 12383748096/134960504832 (9.18%)

Rejected...: 0/12383748096 (0.00%)

Restore.Point...: 153600/1679616 (9.14%)

Restore.Sub.#1...: Salt:0 Amplifier:40704-40960

Iteration:0-256

Candidates.#1...: M1t5I22 -> L4tnsmo

Y cracheando 5.321.783 hashes.

Finalmente utilizamos el diccionario mas grande (Rockyou) con el siguiente comando:

```
hashcat -m 0 hashes.txt rockyou.txt
```

Esto nos tomo aproximadamente 20 min y cracheo finalmente 6.042.717 de hashes. el 36.55% del total en solo 20 min.

Session...: hashcat

Status...: Exhausted

Hash.Name...: MD5

Hash.Target...: hashes.txt

Time.Started...: Wed Sep 9 16:31:02 2020 (27 secs)

Time.Estimated...: Wed Sep 9 16:31:29 2020 (0 secs)

Guess.Base...: File (rockyou.txt)

Guess.Queue...: 1/1 (100.00%)

Speed.#1...: 323.0 kH/s (1.22ms) @ Accel:1024 Loops:1 Thr:1 Vec:8

Recovered...: 6042717/16532955 (36.55%) Digests

Remaining...: 10490238 (63.45%) Digests

Recovered/Time...: CUR:N/A,N/A,N/A AVG:264743,15884594,381230255 (Min,Hour,Day)

Progress...: 14344384/14344384 (100.00%)

Rejected...: 0/14344384 (0.00%)

Restore.Point...: 14344384/14344384 (100.00%)

Restore.Sub.#1...: Salt:0 Amplifier:0-1 Iteration:0-1

Candidates.#1...: \$HEX[206b6d3831303838] -> \$HEX

[042a0337c2a156616d6f732103]

Como conclusión de este ejercicio vemos como tener un buen diccionario es la mejor forma de crackear hashes en cuanto a tiempo y cantidad de hashes crackeados.