# Seguridad Ofensiva 2020: Trabajo Práctico 1

Federico Juan Badaloni y Damián Ariel Marotte

14 de septiembre de 2020

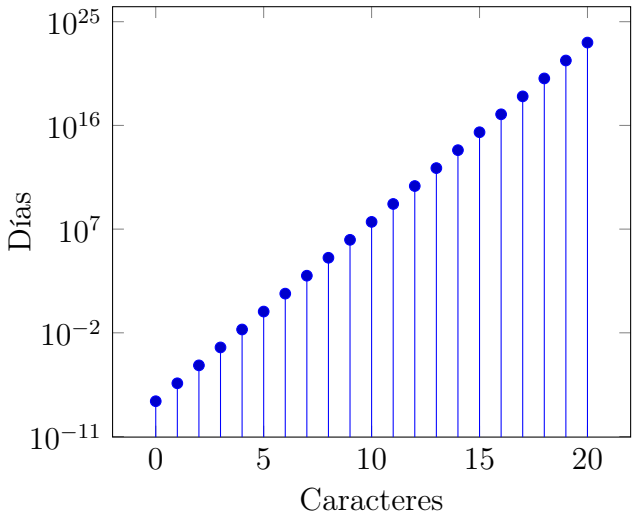
# Ejercicio 1

- Dado un conjunto de $n$ caracteres existen $n^m$ palabras de longitud $m$, por lo tanto para un juego de 36 caracteres y $n \leq 20$ existe un total de

$$\sum_{i=0}^{20} 36^i$$

palabras.

- El siguiente gráfico muestra cuantos días se demora en generar palabras de un largo especifico:




Tenga presente al analizar dicha gráfica, la escala logarítmica del eje vertical.

- Puede utilizarse este script de Python 3 para generar palabras de largo $n$:

```
from itertools import product
import argparse

letras = "abcdefghijklmnopqrstuvwxyz0123456789"
```

```
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument("n", type = int)
    args = parser.parse_args()

    for palabra in product(letras, repeat = args.n):
        print("".join(palabra))
```

Si se desea obtener un archivo, debe redirigirse la salida estándar, mediante el comando `python palabras_largo_n.py n > archivo.txt`.

# Ejercicio 2

Para poder crackear un hash desconocido debemos averiguar con que algoritmo fue generado.

Para ello utilizaremos el siguiente script de bash, que forzará a `hashcat` a intentar crackear el hash que se pasa como argumento, con todos los algoritmos.

```
#!/bin/bash

for i in $(./hashcat -h | grep MD4 -A212 | cut -d'|' -f1 | sort | uniq); do
    ./hashcat -m $i -a 3 --custom-charset1=0 $1 ?1 &> /dev/null
    if (($? == 1)); then
        echo $i
    fi
done
```

La variable de entorno `$?` almacena el valor de retorno del ultimo comando, y es utilizada por el script para determinar si el hash se corresponde con el algoritmo actual.

Para el hash del ejercicio, el script produce como salida los códigos 100, 18500, 300, 4500, 4700 y 6000. Con dicha información puede realizarse el ataque adecuado.

# Ejercicio 3

Para encontrar que palabras del diccionario `ekoparty.txt` que no se encuentran en el diccionario `Passwords.txt` puede utilizarse el siguiente comando:

```
comm -23 ekoparty.txt Passwords.txt
```

pero debe tenerse en cuenta que `comm` asume que ambos diccionarios están ordenados.

Para generar un diccionario con todo el contenido del directorio `Passwords` debe utilizarse el siguiente comando:

```
sort *.txt | uniq > Passwords.txt
```

Las siguiente personas no se encontraron en el directorio `Passwords` ni en las listas mas grandes de http://ns2.elhacker.net/wordlists/:

*big_ b*
*El-Ju3z*
*FiReRaIN*
*GiBa*
*LEX_ LUTOR*
*nicky_ ramone*
*Phreaking Arpac Eniac de Delphi*
*PoCaPiLa*
*QQmelo*
*RaXr*
*RicoTuco*
*SiS_ d0wn*

# Ejercicio 4

- Analizaremos la distribución de frecuencias de longitudes para algunos diccionarios.
  - La utilidad `pipal` arroja los siguientes resultados:

| `darkweb2017-top10000` | `probable-v2-top12000` | `xato-net-100000` |
|---|---|---|
| 6 = 2896 (28.96 %) | 6 = 3611 (28.56 %) | 8 = 34179 (34.18 %) |
| 7 = 2328 (23.28 %) | 8 = 3416 (27.01 %) | 6 = 33895 (33.9 %) |
| 8 = 2162 (21.62 %) | 7 = 2537 (20.06 %) | 7 = 13850 (13.85 %) |
| 9 = 985 (9.85 %) | 5 = 1016 (8.03 %) | 4 = 6910 (6.91 %) |
| 10 = 684 (6.84 %) | 9 = 882 (6.98 %) | 5 = 5938 (5.94 %) |
| 5 = 395 (3.95 %) | 4 = 585 (4.63 %) | 9 = 2807 (2.81 %) |
| 4 = 273 (2.73 %) | 10 = 364 (2.88 %) | 10 = 1405 (1.41 %) |
| 3 = 64 (0.64 %) | 11 = 92 (0.73 %) | 11 = 450 (0.45 %) |
| 11 = 53 (0.53 %) | 3 = 68 (0.54 %) | 12 = 310 (0.31 %) |
| 12 = 46 (0.46 %) | 12 = 40 (0.32 %) | 3 = 76 (0.08 %) |
| 1 = 45 (0.45 %) | 1 = 20 (0.16 %) | 13 = 70 (0.07 %) |
| 2 = 20 (0.2 %) | 2 = 8 (0.06 %) | 14 = 36 (0.04 %) |
| 13 = 10 (0.1 %) | 13 = 6 (0.05 %) | 15 = 30 (0.03 %) |
| 14 = 8 (0.08 %) | | 16 = 22 (0.02 %) |
| 32 = 6 (0.06 %) | | 17 = 6 (0.01 %) |
| 18 = 5 (0.05 %) | | 20 = 6 (0.01 %) |
| 16 = 4 (0.04 %) | | 18 = 5 (0.01 %) |
| 20 = 2 (0.02 %) | | 19 = 3 (0.0 %) |
| 15 = 2 (0.02 %) | | 1 = 1 (0.0 %) |
| 30 = 2 (0.02 %) | | |
| 38 = 1 (0.01 %) | | |
| 21 = 1 (0.01 %) | | |
| 36 = 1 (0.01 %) | | |
| 33 = 1 (0.01 %) | | |
| 24 = 1 (0.01 %) | | |
| 17 = 1 (0.01 %) | | |
| 26 = 1 (0.01 %) | | |
| 27 = 1 (0.01 %) | | |
| 19 = 1 (0.01 %) | | |

- Podemos obtener resultados similares utilizando el comando

```
awk '{ print length($0) }' archivo | sort | uniq -c | sort -rg
```

el cual arroja la siguiente salida:

| darkweb2017-top10000 | probable-v2-top12000 | xato-net-100000 |
|---|---|---|
| 2896 6 | 3611 6 | 34179 8 |
| 2328 7 | 3416 8 | 33895 6 |
| 2162 8 | 2537 7 | 13850 7 |
| 985 9 | 1016 5 | 6910 4 |
| 684 10 | 882 9 | 5938 5 |
| 395 5 | 585 4 | 2807 9 |
| 273 4 | 364 10 | 1405 10 |
| 64 3 | 92 11 | 450 11 |
| 53 11 | 68 3 | 310 12 |
| 46 12 | 40 12 | 76 3 |
| 45 1 | 20 1 | 70 13 |
| 20 2 | 8 2 | 36 14 |
| 10 13 | 6 13 | 30 15 |
| 8 14 | | 22 16 |
| 6 32 | | 6 20 |
| 5 18 | | 6 17 |
| 4 16 | | 5 18 |
| 2 30 | | 3 19 |
| 2 20 | | 1 1 |
| 2 15 | | 1 0 |
| 1 38 | | |
| 1 36 | | |
| 1 33 | | |
| 1 27 | | |
| 1 26 | | |
| 1 24 | | |
| 1 21 | | |
| 1 19 | | |
| 1 17 | | |

- A continuación, un listado de los diez sufijos numéricos mas utilizados.

  - Según `pipal`:

| darkweb2017-top10000 | probable-v2-top12000 | xato-net-100000 |
|---|---|---|
| 3456 = 52 (0.52 %) | 1234 = 24 (0.19 %) | 2000 = 418 (0.42 %) |
| 1234 = 40 (0.4 %) | 2345 = 14 (0.11 %) | 1995 = 396 (0.4 %) |
| 2345 = 38 (0.38 %) | 4321 = 13 (0.1 %) | 1996 = 394 (0.39 %) |
| 4321 = 18 (0.18 %) | 3456 = 11 (0.09 %) | 1987 = 392 (0.39 %) |
| 1111 = 17 (0.17 %) | 0000 = 9 (0.07 %) | 1994 = 391 (0.39 %) |
| 0000 = 13 (0.13 %) | 1111 = 9 (0.07 %) | 1988 = 391 (0.39 %) |
| 6789 = 13 (0.13 %) | 2222 = 7 (0.06 %) | 1985 = 390 (0.39 %) |
| 1314 = 9 (0.09 %) | 9999 = 7 (0.06 %) | 1983 = 388 (0.39 %) |
| 6666 = 8 (0.08 %) | 5555 = 6 (0.05 %) | 1982 = 387 (0.39 %) |
| 9999 = 7 (0.07 %) | 3333 = 5 (0.04 %) | 1992 = 387 (0.39 %) |

  - También puede obtenerse dicha información ejecutando el comando

    ```
    grep -o '[0-9][0-9][0-9][0-9]$' archivo | sort | uniq -c | sort -rg | head
    ```

    que arroja los siguientes resultados:

| darkweb2017-top10000 | probable-v2-top12000 | xato-net-100000 |
|---|---|---|
| 52 3456 | 24 1234 | 418 2000 |
| 40 1234 | 14 2345 | 396 1995 |
| 38 2345 | 13 4321 | 394 1996 |
| 18 4321 | 11 3456 | 392 1987 |
| 17 1111 | 9 1111 | 391 1994 |
| 13 6789 | 9 0000 | 391 1988 |
| 13 0000 | 7 9999 | 390 1985 |
| 9 1314 | 7 2222 | 388 1983 |
| 8 6666 | 6 5555 | 387 1992 |
| 7 9999 | 5 8888 | 387 1982 |

# Ejercicio 5

El siguiente programa, se encarga de calcular los datos necesarios para recuperar los mensajes:

```
# Datos
N1 = 389573830229905951812919842231016962853053655719189056621093978169837233625740
N2 = 303668390381967550574109116494546194718900491649463376637217628240940969495870
N3 = 479345567729954913738228458501575007323911241436168052925595131821796030084130
C1 = 396708474546125804352894757436683688457291028695044217325853929491171136935480
C2 = 355006513750551550798931713354683491263062473879176656452255051848683497534660
C3 = 924835278307680480966328618545268895077532556525413716080960421925985654497130

def egcd(a, b):
    if a == 0:
        return (b, 0, 1)
    else:
        g, y, x = egcd(b % a, a)
        return (g, x - (b // a) * y, y)

def modinv(a, m):
    g, x, y = egcd(a, m)
    if g != 1:
        raise Exception('modular inverse does not exist')
    else:
        return x % m

# Calculamos los primos como los maximos comunes divisores entre los N
Q = egcd(N1, N2)[0]
R = egcd(N2, N3)[0]
P = egcd(N3, N1)[0]

# Calculamos los phi
phi1 = (P - 1) * (Q - 1)
phi2 = (Q - 1) * (R - 1)
phi3 = (P - 1) * (R - 1)
```

```
# Los d se pueden calcular como inversos modulares de e con los distintos phi
d1 = modinv(E, phi1)
d2 = modinv(E, phi2)
d3 = modinv(E, phi3)

# d y N completan la clave privada.
# Podemos aplicar la formula de RSA para desencriptar los textos planos.
pt1 = pow(C1, d1, N1)
pt2 = pow(C2, d2, N2)
pt3 = pow(C3, d3, N3)
print(f"pt1: {pt1}")
print(f"pt2: {pt2}")
print(f"pt3: {pt3}")

# Podemos encriptar de nuevo para verificar que los resultados son correctos
assert pow(pt1, E, N1) == C1
assert pow(pt2, E, N2) == C2
assert pow(pt3, E, N3) == C3
```

Al finalizar el programa se muestran por pantalla los mensajes recuperados: 53181249696568492218371956806067620 8, 26682430532099256916006624488943857 9 y 54193405533754713988023425573572646 1.

# Ejercicio 6

a) Luego de observar el comportamiento del servidor para passwords de una sola letra, notamos que procesar la letra «G» lleva mas tiempo que las demás.

   Esto nos lleva a pensar que las contraseñas se procesan letra a letra, y que el procedimiento mediante el cual se rechaza una letra es notablemente mas rápido que el que se emplea para aceptarla.

   Con esta información en mente, se desarrollo en siguiente script, que puede obtener la contraseña («*GaAVCK9r3K*») hasta en menos de 50 segundos:

```
#!/bin/bash

PASS=""
rm /tmp/pass -f

while true; do
  echo Cracking... $PASS

  for i in {{A..Z},{a..z},{0..9}}; do
    touch /tmp/pass.$i
    (echo $PASS$i | nc 143.0.100.198 60123 &>> /tmp/pass ; rm /tmp/pass.$i -f)&
  done

  until [ $(ls /tmp/pass.* 2> /dev/null | wc -l) == "1" ]; do
    if grep -qs Felicitaciones /tmp/pass; then
      rm /tmp/pass.* -f
      echo Password: $PASS
      exit 0
    fi
  done

  PASS="${PASS}$(ls /tmp/pass.* 2> /dev/null | cut -d. -f2)"
done
```

Tenga en cuenta que si el servidor se satura y las respuestas incorrectas se demoran, entonces será imposible distinguir los aciertos de los errores y no se obtendrá un resultado correcto.

b) Sea $t$ el tiempo que se tarda en aceptar una letra, $l$ la longitud de la contraseña y asumiendo que rechazar una letra no produce retraso; paralelizando los procesos puede esperarse obtener el resultado en:

$$\sum_{i=0}^{l-1} it + t$$

# Ejercicio 7

Se lograron crackear mas de 16.500.000 de contraseñas. Para realizar los diferentes ataques se tuvieron en cuenta (entre otras) las listas de palabras que se encuentran en:

- https://weakpass.com
- https://leakhispano.net/
- https://github.com/danielmiessler/SecLists
- http://ns2.elhacker.net/wordlists/
- https://wiki.skullsecurity.org/Passwords

Se llevaron a cabo principalmente tres tipos de ataque:

- Ataques por mascaras con el comando

  ```
  ./hashcat -a 3 -m 0 hashes.txt masks/rockyou-7-2592000.hcmask
  ```

- Ataques de diccionarios con el comando

  ```
  ./hashcat -m 0 hashes.txt diccionario
  ```

- Ataques de producto cartesiano con el comando

  ```
  ./hashcat -a 1 -m 0 hashes.txt diccionario1 diccionario2
  ```

Los resultados de estos ataques pueden observarse en cualquiera de los siguientes mirrors:

- Google Drive
- Mega
- One Drive
- Dropbox
- Github

Los 25 passwords mas utilizados en 2019 (según SplashData) fueron todos encontrados.