03 de Marzo del 2024

Explicación técnica: Explotación de vulnerabilidad *export-grade* en protocolo para intercambio de llaves Diffie-Hellman

ITESO, Universidad Jesuita de Guadalajara

Autor:

- José Manuel Rodríguez Vázquez – 746193

Hipótesis

El problema que existe en el intercambio de claves Diffie-Hellman, utilizado en la criptografía representa un problema bastante grave, varias técnicas a probar como Fuerza bruta, LogJam, Baby steps -> Giant steps y Pollard's Rho, se pueden utilizar para vulnerar este protocolo de intercambio de llaves, las hipótesis relacionadas a las técnicas con las siguientes:

- **Fuerza bruta:** Con la evolución del poder computacional a lo largo del tiempo, un ataque de fuerza bruta se vuelve cada vez más viable para romper estos protocolos.
- **LogJam:** Al bajar de nivel la negociación de clave, podemos utilizar de forma más sencilla el problema de logaritmo discreto para obtener el exponente.
- **Baby steps -> Giant steps:** Este algoritmo tiene una complejidad temporal $O(\sqrt{p})$, lo que puede hacer más eficiente el proceso de un ataque de fuerza bruta.
- **Pollard 's Rho:** Puede resultar más eficiente debido a la aleatoriedad que maneja, aunque, puede durar mucho en ciertas ocasiones.

Métodos y algoritmos empleados durante el proyecto

1- Fuerza bruta

Método bastante sencillo de aplicar, el cual tuvo como base la iteración entre un rango $[1, p-1]$, intercalando los valores de este por el exponente secreto x que genera A, de tal manera que [$g^x \ mod \ p \ = \ A$]

Una implementación básica de este algoritmo logró expresarse de la siguiente manera:

```
p = "0xde26ab651b92a129"
g = 2
A = str(input("Intercepted A: "))

p_i=int(p, 16)
A_i=int(A, 16)

B = str(input("Intercepted B: "))
B_i=int(B, 16)

for i in range(1, p-1):
        secret1=pow(g, i, p_i)
        if secret1==A_i:
                print("[+] Clave conjunta encontrada")
                secret=i
                break
if secret is None:
        print("[!] Clave conjunta no encontrada")

shared_secret=pow(B_i, secret, p_i)
```

2- Baby steps → Giant steps

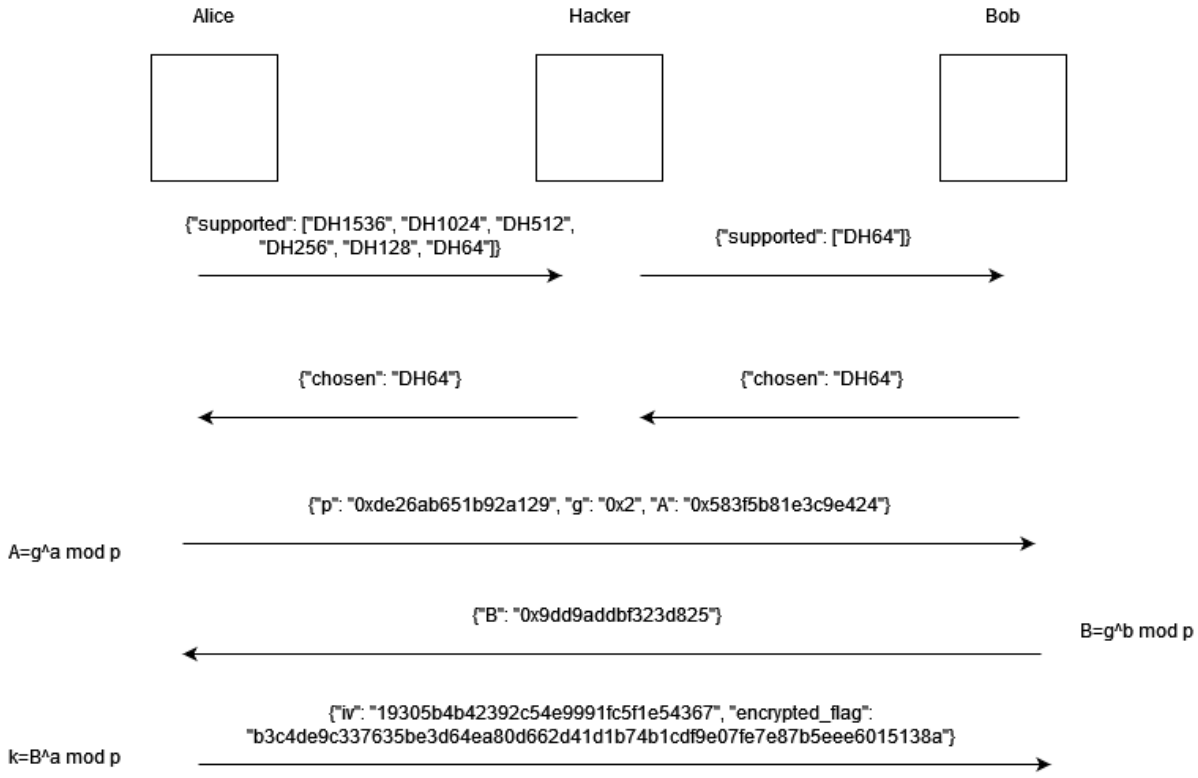
Algoritmo basado en la implementación de una lista, arreglo, etc, la cual aumenta su tamaño y valores dentro a través de iteraciones [$1, \sqrt{p}$] (Baby steps), aumentando su eficiencia al reducir considerablemente el rango de búsqueda.
Estos valores almacenados, a su vez son comparados por un valor que crece de manera exponencial (Giant Steps) a través de la fórmula [$valor = (valor * g) \, mod \, p$]. Cuando los valores coinciden, se habrá encontrado el exponente x tal que [$g^x \, mod \, p = A$].

Al trabajar con un número entero de 64 bits, la lista empleada crece de manera exponencial, por lo que se optó por almacenar estos valores directamente en un archivo dentro del almacenamiento físico, en lugar de dejarlos en la memoria RAM.
Se empleó de la siguiente manera:

```python
def baby_steps(p, g, A, root_p):
    value = 1
    baby_steps_dict = {}
    with open("baby_steps.txt", "w") as f:
        for x in range(root_p):
            f.write(f"{value} {x}\n")
            value = (value * g) % p

def gigant_steps(p, g, A, root_p):
    gigant_steps = pow(pow(g, root_p, p), -1, p)
    value = A
    with open("baby_steps.txt", "r") as f:
        for line in f:
            baby_value, baby_x = map(int, line.strip().split())
            if value == baby_value:
                return int(baby_x)
        value = (value * gigant_steps) % p
    return -1

A_i = int("0x5cda3a2dc931af7e", 16)
p_i = int("0xde26ab651b92a129", 16)
g = 2
B_i = int("0x2e9ebd2b01b84a78", 16)

print("[-] Calculando raiz de p...")
root_p = int(math.ceil(math.sqrt(p_i - 1)))
print("[+] Raiz de p encontrada: ", root_p)

secret = gigant_steps(p_i, g, A_i, root_p)
print(f"Secret found: {secret}")
```

3- Logaritmo discreto

Este ataque utiliza logaritmo, discreto para poder encontrar el exponente y con ello poder conseguir la llave que está siendo pasada mediante Diffie-Hellman, gráficamente podemos ver que el ataque consiste en:



Lo que como MITM logramos interceptar es simplemente elegir la versión de Diffie-Hellman a utilizar, elegimos la más débil y con ello podemos lograr solamente ver lo que se manda, así que debemos obtener el exponente, para ello usamos sencillamente las matemáticas:

$$A = g^x mod\ p$$

Por lo tanto realizamos el despeje con logaritmo:

$$x = log_g A\ mod\ p$$

Para ello utilizamos el siguiente código haciendo uso de la librería “sympy”, que lleva a cabo tareas de álgebra computacional, el código queda de la siguiente forma:

```
import math
```

```python
import hashlib
from Crypto.Cipher import AES
from Crypto.Util.Padding import unpad
from sympy.ntheory import discrete_log

encrypted_flag="aed26e89092693ee4266b676e1eac3a5805df0e34ab3b17430b1e6ada72754df"
phex="0xde26ab651b92a129"
ghex="0x2"
Ahex="0xb727cdc5e6c9a262"
Bhex="0x8cf4521fba2fd1ea"
iv=bytes.fromhex("c62454a245526136d41557b035310700")
p = int(phex, 16)
g = int(ghex, 16)
A = int(Ahex, 16)
B = int(Bhex, 16)
encrypted_flag = bytes.fromhex(encrypted_flag)

def decrypt_flag(shared_secret: int, iv: str, ciphertext: str):
    sha1 = hashlib.sha1()
    sha1.update(str(shared_secret).encode('ascii'))
    key = sha1.digest()[:16]
    aes = AES.new(key, AES.MODE_CBC, iv)
    plaintext = aes.decrypt(ciphertext)
    return plaintext

def DLP(g, A, p):
    return discrete_log(p, A, g)

a = DLP(g, A, p)
shared_secret=pow(B,a,p)
text = decrypt_flag(shared_secret,iv,encrypted_flag)
print(text)
```

4- Algoritmo de enfoque aleatorio

Este algoritmo es el más complejo que se trató en la investigación. Se trató de una mezcla entre el algoritmo Rho de Pollard y el Algoritmo de Richard P. Brent para la detección de ciclos. En este planteamiento nos centramos en ***dos elementos*** que partirán desde la misma posición hasta que en algún punto aleatorio, colisionen para reducir los rangos de búsqueda Este algoritmo se puede dividir en cinco etapas:

1. ***Pocisionamiento de origen:***
   En esta fase, se asignan dos números aleatorios en representación de A y B (a_rand, b_rand respectivamente) que varían en un rango de $[1, p-1]$. Estos números cumplen la función para conformar dos conjuntos de tres partes (y, a, b), las cuales son:

$$y = (((g^{a\ rand}) \bmod p) * ((A^{b\ rand}) \bmod p)) \bmod p$$
$$\boldsymbol{a} = a\_rand$$
$$\boldsymbol{b} = b\_rand$$

   Si **'y'** cumple ciertos requisitos dentro del ciclo de colisión, alterará su valor, al igual que el de **a** y **b** respectivamente.
   A ambos conjuntos con el mismo valor inicial, les llamaremos **tortuga** y **liebre**

2. ***Ciclo de colisión:***
   Durante esta iteración, se modificarán los valores de (y, a, b) dependiendo del resultado de $y \bmod p$. Se elige el número 3 en este algoritmo debido a que es un número chico que simplifica los cálculos, pero podría ser cualquier primo que eligiéramos.

Si $y \bmod 3 = 0$:
$$y = (g * y) \bmod p$$
$$a = a + 1$$

Si $y \bmod 3 = 0$:
$$y = (A * y) \bmod p$$
$$b = b + 1$$

Cualquier otro residuo resultará en:
$$y = (y^2) \bmod p$$
$$a = a * 2$$
$$b = b * 2$$

   Por lógica de aleatoriedad, en algún punto la **'y'** de ambos elementos **tortuga** y **liebre**. serán iguales.

3. ***Reasignación de valores iniciales post-bucle:***
   Se toman en segundo y tercer valor de ambos elementos de la siguiente manera:

Tortuga: aT (segundo valor), bT (tercer valor)
Liebre: aL (segundo valor), bT (tercer valor)

Con estos valores, recalcularemos a y b nuevamente:

$$a = aL - aT$$
$$b = bT - bL$$

4. ***Reducción de valores***
   En este punto del algoritmo, se inicializa un valor **'d'**, generado a través del Máximo Común Divisor entre $\boldsymbol{bL - bT}$ y $p - 1$.
   Este MCD es el valor real de **d**, con el cual normalizamos los valores **a, b** a un rango más corto
   Los valores normalizados de a,b se calculan con el valor entero de la división entre **d**, de la siguiente manera:

$$a = a//d$$
$$b = b//d$$

   De igual manera, normalizamos el módulo actual (p), para reducir todos los valores posibles. Este se calculará como

$$n\_mod = p - 1 \,//\, d$$

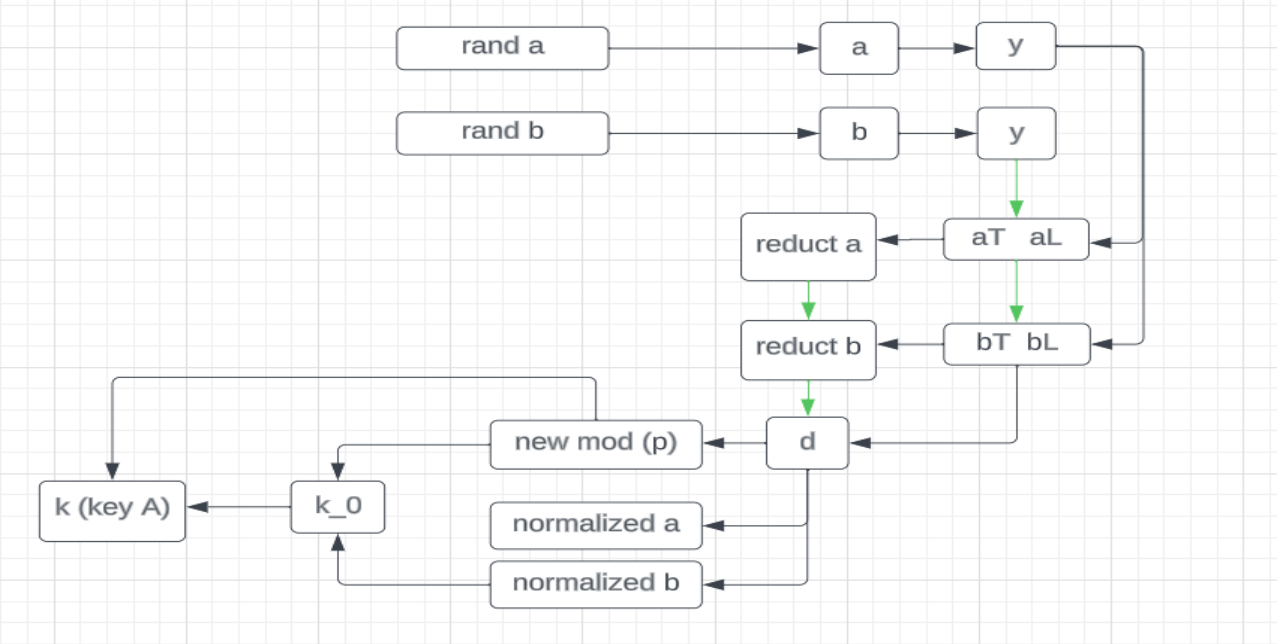
5. **Calcular candidatos de k que genere A**
   Generamos una k_0 a partir del inverso modular de b en el nuevo módulo, aplicando al inverso de b el módulo nuevamente
   $k_0 = (b^{-1} \, mod \, n\_mod) \, mod \, n\_mod$

   Y con k sub-zero calculada, iteramos aumentando el valor de n_mod una vez por cada ciclo, es decir, calculamos k aumentando el valor de k_0 en $n + 1 \, * n\_mod$ pasos

$$k = k_0 \, + \, (n + 1) \, * \, n\_mod$$

Por lo que, dependiendo del número par en el que partamos, generamos más o menos rápido el valor que genera A.

De manera resumida, los valores se generan de la siguiente manera:



El código del algoritmo quedaría planteado de la siguiente manera:

```
A = "0x5cda3a2dc931af7e"
p = "0xde26ab651b92a129"
g = 2
B = "0x2e9ebd2b01b84a78"
p = int(p, 16)
A = int(A, 16)
B = int(B, 16)

print(f'\t--> Parametros: (p, g, A) = ({p}, {g}, {A})')

def f(runner):
    y, a, b = runner
    if y%3 == 0:
        y = g*y %p
        a += 1
    elif y%3 == 1:
        y = A*y %p
        b += 1
    else:
        y = y*y %p
        a *= 2
        b *= 2
    return y, a%(p-1), b%(p-1)

a_rand = random.randint(1, p-2)
b_rand = random.randint(1, p-2)
liebre = tortuga = (pow(g, a_rand, p)*pow(A, b_rand, p) % p, a_rand, b_rand)

while True:
    tortuga = f(tortuga)
    liebre = f(f(liebre))
     if tortuga[0] == liebre[0]:
```

```
        break

aT, bT = tortuga[1:]
aL, bL = liebre[1:]
a = aL - aT
b = bT - bL
d = math.gcd(bL-bT, p-1) #mcd entre ambos valores, para posible d

a = a//d #Normalizar valor a
b = b//d #Normalizar valor b
new_mod = (p-1)//d #Normalizar nuevo modulo

k0 = a * pow(b, -1, new_mod) % new_mod
print(f'[-] Revisando posible candidato para la clave d = {d}')

for _ in range(d): # _ es d-1
    k = k0 + _ * new_mod # calcula k desde el inicio k0 + d-1 + el nuevo primo
    print(f'\t--> For k = {k}, g^k == {pow(g,k,p)} modulo {p}.') # traza
    if pow(g,k,p) == A: #Vemos que la clave funcione correctamente para generar A
        print(f'[+] AVISO: Clave compartida encontrada: {k}')
        break
```

Resultados

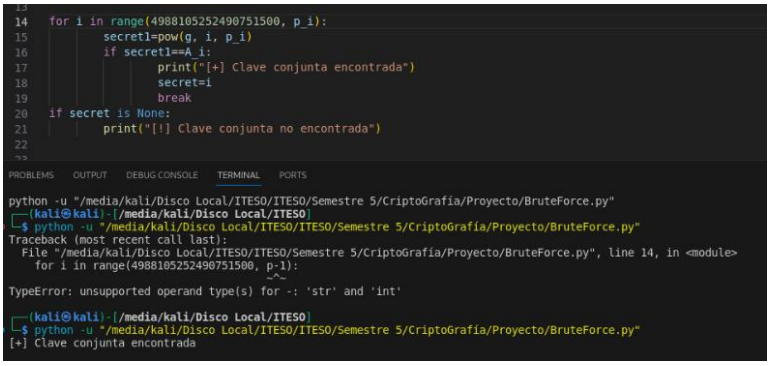
Para automatizar los resultados que funcionaron, utilizamos un script en bash para automatizar las conexiones y pasar las variables al código de python, se logró de la siguiente manera:

```
#!/bin/bash
function ctrl_c() {
  echo -e "${red}\n\n[!] Saliendo...\n${normal}"
  rm $py_file 2>/dev/null
  rm out.txt 2>/dev/null
  exit 1
}
# Ctrl+c
trap ctrl_c INT

red='\033[1;31m'
gray='\033[1;30m'
normal='\033[0m'
########## Banners ##########
py_file=LogJam.py
target="socket.cryptohack.org 13379"

echo -ne "[+] Estableciendo conexión a ${target}"
echo
tempfile=$(mktemp)
{
  echo -ne '{"supported": ["DH64"]}\n';
  sleep 2;
  echo -ne '{"chosen": "DH64"}\n';
  sleep 2;
} | nc $target >$tempfile &
nc_pid=$!
received_data=false
wait $nc_pid

> out.txt

while read -r line; do
  echo "$line" | tee -a out.txt
  if [[ "$line" == *'"encrypted_flag"'* ]]; then
    received_data=true
    break
  fi
done <$tempfile

rm $tempfile

if $received_data; then
  echo -ne "\n[+] Comunicación terminada" | tee -a out.txt
else
  echo "[!] No se recibieron datos"
fi
#Valores
A=$(cat out.txt | grep -o '"A": "0x[^"]*' | cut -d'"' -f4)
iv=$(cat out.txt | grep -o '"iv": "[^"]*' | awk -F ': ' '{print $2}' | tr -d '"')
encrypted_flag=$(cat out.txt | grep -o '"encrypted_flag": "[^"]*' | awk -F ': ' '{print $2}' | tr -d '"')
B=$(cat out.txt | grep -o '"B": "0x[^"]*' | cut -d'"' -f4)
p=$(cat out.txt | grep -o '"p": "0x[^"]*' | cut -d'"' -f4)
```

```
py_load=$(cat << EOL
)
######################## Código de python con variables de bash ########################
EOL
)
echo "$py_load" > "$py_file"
python3 "$py_file"
rm "$py_file"
rm out.txt
```

- **Fuerza bruta:**

Este método deja de ser óptimo, ya que tendríamos que disponer de por lo menos 1 millón de años, por lo que nos vimos obligados a buscar otros métodos para “atacar” este problema. Sin embargo, comprobamos que funciona al poner un rango cercano que ya conocíamos, y encontrar la clave generadora de A.



- **Logaritmo:**

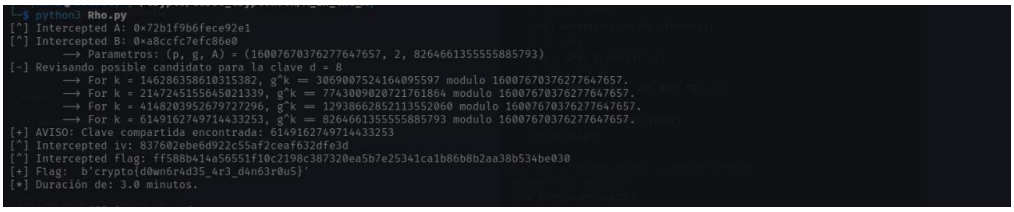
Es un método de “ataque” a AES, por lo que
Se llevó a cabo la automatización del script para que todas las llaves y cosas a interceptar se hicieran automáticamente, para ello se hizo uso de una combinación de bash y python.

```
[+] Estableciendo conexión a socket.cryptohack.org 13379
Intercepted from Alice: {"supported": ["DH1536", "DH1024", "DH512", "DH256", "DH128", "DH64"
]}
Send to Bob: Intercepted from Bob: {"chosen": "DH64"}
Send to Alice: Intercepted from Alice: {"p": "0xde26ab651b92a129", "g": "0x2", "A": "0xafcbc
3ea854ec034"}
Intercepted from Bob: {"B": "0xd1ebb2dc8f736c81"}
Intercepted from Alice: {"iv": "960506e6eff82bde5f1659104a51acf9", "encrypted_flag": "a6871a
4ee1da52f4f4430e1ab6d9fba10a87b7a54f285ed8f97d72438f58a652"}

[+] Comunicación terminada
Base:  2
Power:  12667433788935946292
Modulus:  16007670376277647657
Alice's secret key:  7825658248020108114
Flag:  b'crypto{d0wn6r4d35_4r3_d4n63r0u5}'
```

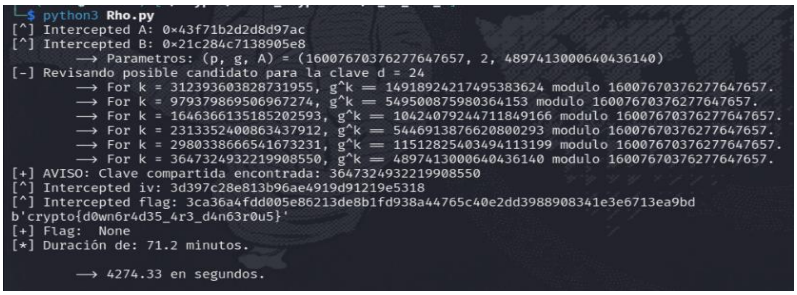
- **Ataque de semilla aleatoria:**

Se demostró que la aleatoriedad del programa puede ser tanto beneficiosa como problemática, al rondar tiempos de ejecución que podían ir desde los 3 minutos hasta las 2 horas. Pero siempre oscilando en un promedio de entre 30 a 40 minutos, tanto para 64 como para 128 bits:

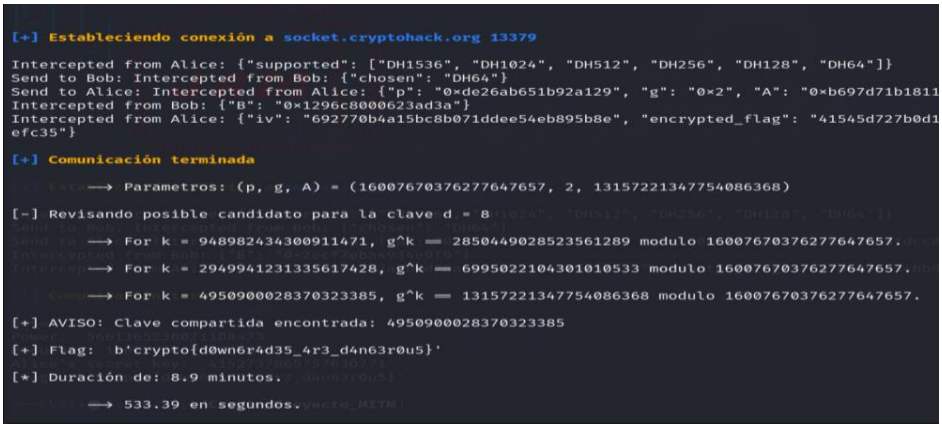
- 3 minutos con 64 bits:



- 71 minutos con 64 bits:



- 8 minutos con 128 bits:

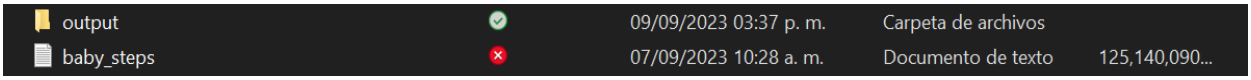


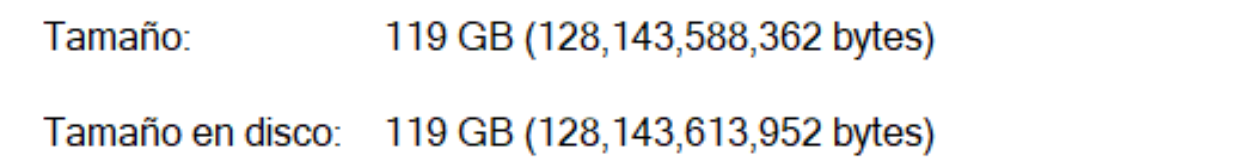
- **Baby steps → Giant steps**

Con este algoritmo, es posible resolver el problema planteado; se llegó a la conclusión de que no es rentable con el hardware que disponemos, puesto que por cada nueva ejecución con

nuevos valores, se requerirían 128Gb disponibles, ya fuese de memoria RAM o de almacenamiento físico.
Sin mencionar que, a pesar de contar con una lista Arcoiris previamente generada esta toma una excesiva cantidad de tiempo en iterar. Aun con una eficiencia de $O\sqrt{p}$, es bastante menos eficiente que otros algoritmos.

Aquí podemos observar la cantidad de espacio que ocupa la lista Arcoiris generada en un archivo de texto, para un número de 64 bits. Podemos observar que el archivo supera los 100Gb de almacenamiento:





Referencias extra:

Conrad, S. (2015, May 22). Logjam attack, similar to the FREAK vulnerability, breaks TLS security. Emsisoft | Cybersecurity Blog. https://www.emsisoft.com/en/blog/16757/logjam-attack-similar-to-the-freak-vulnerability-breaks-tls-security/#:~:text=The%20Logjam%20attack%20technique%20involves,passing%20through%20the%20affected%20connection

Paar, C., & Pelzl, J. (2009). Understanding Cryptography: A Textbook for students and practitioners. http://euro.ecom.cmu.edu/resources/elibrary/epay/Sigs.pdf

[Logjam Vulnerability] Explanation and Prevention Guide. (2021, April 2). https://crashtest-security.com/logjam-tls/

Pollard's Rho Algorithm for Prime Factorization. (2016, January 3). GeeksforGeeks. https://www.geeksforgeeks.org/pollards-rho-algorithm-prime-factorization/