Memoria Laboratorio 2: Ataque de Fuerza Bruta en GPG

Universidad de Castilla La-Mancha. Escuela Superior de Informática

(42340) Seguridad en Redes – José Luis Segura Lucas

Autor: Pedro José Medina Sánchez

1. Introducción

Para esta práctica, he decidido llevar a cabo un ataque de fuerza bruta sobre un archivo cifrado con GPG. El objetivo es encontrar la contraseña correcta para descifrar dicho archivo, teniendo en cuenta que la clave contiene solo letras minúsculas. Aunque también consideré la opción de realizar un ataque por diccionario, que sería más eficiente si tuviera una lista de posibles contraseñas, opté por el ataque de fuerza bruta debido a la falta de información sobre la clave.

2. Elección del lenguaje de programación

He elegido Python como el lenguaje de programación para realizar esta práctica por varias razones:

- **Facilidad para generar scripts:** Python tiene una sintaxis sencilla y limpia, lo que permite desarrollar scripts de forma rápida y eficiente.
- **Manejo de hilos y procesos:** Python ofrece herramientas muy útiles para manejar tanto multihilos (threading) como multiprocesos (multiprocessing), lo cual facilita el desarrollo de soluciones que aprovechen al máximo el hardware disponible.
- **Compatibilidad con GPG:** Además, Python dispone de varias librerías para interactuar con GPG, como gnupg y subprocess, lo que me permitió probar diferentes enfoques para realizar el ataque.

3. Entorno controlado de pruebas

Para garantizar que los resultados obtenidos en las pruebas de los distintos scripts fueran lo más consistentes y fiables posible, he decidido utilizar un entorno de pruebas controlado. Esto es fundamental para asegurar que todas las implementaciones se ejecutan bajo las mismas condiciones y en el mismo hardware, de manera que las diferencias en los tiempos de ejecución se deban únicamente a la eficiencia del código y no a factores externos como la variación en el uso de la CPU o la interferencia de otros procesos del sistema.

El entorno controlado que he utilizado es un contenedor Docker, lo cual me ha permitido crear un sistema operativo limpio y aislado, con los recursos asignados específicamente para la ejecución de los scripts. De esta forma, me aseguro de que todos los scripts se ejecutan en igualdad de condiciones en cuanto a la disponibilidad de núcleos de CPU, RAM, y sistema operativo.

El entorno de pruebas utiliza Ubuntu 20.04 como imagen base, sobre la cual se han instalado las dependencias necesarias para ejecutar los scripts de Python que interactúan con GPG. También se copia al contenedor el archivo cifrado y los scripts de ataque de fuerza bruta para ejecutarlos dentro del contenedor.

4. Desarrollo

A lo largo de la práctica, he implementado varios scripts en Python con distintas aproximaciones para llevar a cabo el ataque de fuerza bruta, aprovechando diferentes formas de ejecutar el código y paralelizarlo. A continuación, describo cada una de las implementaciones:

1. **Ataque secuencial usando subprocess:** El script genera todas las combinaciones posibles de contraseñas utilizando la función *itertools.product*, comenzando desde contraseñas de 1 carácter y aumentando progresivamente la longitud hasta que se encuentra la contraseña correcta o se prueban todas las combinaciones posibles. Cada combinación de contraseña se pasa como argumento a GPG utilizando subprocess.run, que ejecuta el comando *gpg --decrypt* con la contraseña como parámetro. Si GPG devuelve un código de salida de éxito *(returncode == 0)*, el script imprime la contraseña encontrada y termina.

   Dado que este es un enfoque secuencial y utiliza un solo núcleo de la CPU para probar todas las combinaciones, espero que sea uno de los métodos más lentos. Aunque el uso de subprocess permite ejecutar GPG de forma directa y eficiente, el hecho de que el script no paralelice las pruebas y las ejecute una tras otra lo hace menos adecuado para este tipo de tareas intensivas en CPU.

2. **Ataque secuencial usando la librería gnupg**: El script *bf_secuential_gnupg.py* es muy similar al anterior, pero en este caso utilizamos la librería gnupg en lugar de subprocess para interactuar con GPG. El script sigue un enfoque secuencial, generando todas las combinaciones posibles de contraseñas y probándolas una por una. La diferencia principal es que gnupg proporciona una API más "pythonizada" y amigable para manejar GPG, lo que facilita la integración directa con Python. Aquí, el archivo cifrado se abre con Python, y el método *gpg.decrypt_file()* se encarga de probar cada contraseña y verificar si es la correcta.

   Aunque el enfoque es muy similar al del script con subprocess, esde esperar que el rendimiento sea más lento en este caso. Esto se debe a que gnupg utiliza archivos temporales para la comunicación entre Python y GPG, lo que añade una sobrecarga. En contraste, subprocess usa pipes, que son más rápidas y eficientes, ya que pasan los datos directamente entre el programa y el proceso GPG sin tener que escribirlos y leerlos de archivos en disco. Este manejo adicional de archivos temporales, junto con las capas de abstracción de gnupg, aumenta el consumo de recursos y el tiempo de procesamiento. A pesar de ser más lenta, esta librería es útil para aquellos que buscan una integración más directa y sencilla con Python, priorizando la usabilidad sobre la eficiencia.

3. **Ataque multithreading**: El script *bf_multithreading.py* es una mejora con respecto a las versiones secuenciales, ya que utiliza hilos (threads) para intentar paralelizar el ataque de fuerza bruta. En este enfoque, divido el trabajo en varios hilos (en este caso, 4 hilos) que trabajan en paralelo para probar diferentes combinaciones de contraseñas. Cada hilo recibe un subconjunto de contraseñas que probar, lo que debería teóricamente acelerar el proceso al permitir que varias combinaciones se prueben simultáneamente. A medida que cada hilo trabaja, si uno encuentra la contraseña correcta, se activa un evento global (*password_found_event*) que detiene la ejecución de los demás hilos para evitar que sigan trabajando innecesariamente.

   En teoría, este enfoque multihilo debería ser más rápido que las versiones secuenciales, ya que permite probar varias contraseñas al mismo tiempo. Sin embargo, en Python existe un problema importante con el multihilo debido al Global Interpreter Lock (GIL). El GIL es un mecanismo que impide que múltiples hilos ejecuten código Python en paralelo dentro de un mismo proceso. Esto significa que, aunque el código esté escrito para usar hilos, en realidad

solo un hilo puede estar ejecutándose en un momento dado si se está ejecutando código Python puro.

Por esta razón, aunque los hilos pueden gestionar tareas de entrada y salida (E/S) en paralelo, en este caso, que es intensivo en CPU, el GIL limitará el rendimiento de la implementación multihilo. Aunque veremos una mejora leve con respecto a la ejecución secuencial, el paralelismo real no se logrará debido a esta limitación, lo que hará que este enfoque no sea tan eficiente como parece a primera vista, especialmente para tareas intensivas en cálculo como este ataque de fuerza bruta.

4. **Ataque multiprocessing**: El script *bf_multiprocessing.py* cambia de enfoque respecto a los anteriores, utilizando multiprocessing en lugar de multithreading o ejecución secuencial. En este caso, se crean procesos independientes en lugar de hilos, lo que permite ejecutar múltiples instancias de GPG en paralelo, distribuyendo la carga de trabajo entre varios núcleos de la CPU. Cada proceso recibe un conjunto de combinaciones de contraseñas para probar, y todos trabajan simultáneamente.

   Al igual que en el caso multihilo, si uno de los procesos encuentra la contraseña correcta, se activa un evento global (*password_found_event*) que detiene la ejecución de los demás procesos, evitando trabajo innecesario. La principal diferencia aquí es que, al usar procesos en lugar de hilos, cada proceso tiene su propio espacio de memoria y no está limitado por el Global Interpreter Lock (GIL) de Python.

5. **Ataque multiprocessing con afinidad de CPU (CPU affinity):** Este script *bf_multiprocessing_cpu_affinity.py* es una versión mejorada del anterior *bf_multiprocessing.py*, pero con la adición de afinidad de CPU (CPU affinity). En este caso, cada proceso está asignado a un núcleo específico de la CPU, lo que asegura que cada proceso se ejecuta en un núcleo fijo, evitando la migración entre núcleos que podría ocurrir si el sistema operativo redistribuyera las cargas entre los cores.

   En cuanto al funcionamiento, se utiliza la función *os.sched_setaffinity()* para fijar cada proceso a un núcleo específico (determinado por el parámetro *core_id*). De este modo, cada proceso ejecuta su parte del trabajo sin ser migrado de un núcleo a otro, lo cual puede mejorar la eficiencia en ciertas tareas, particularmente en aquellas que son intensivas en CPU, como en este ataque de fuerza bruta.

   La afinidad de CPU debería ofrecer una mejora en el rendimiento en este caso. Como se trata de una tarea intensiva en CPU, fijar cada proceso a un núcleo específico es beneficioso porque:

   - **Reducción de la sobrecarga de cambio de contexto**: Si los procesos no están fijados a un núcleo, el sistema operativo puede migrarlos de un núcleo a otro dependiendo de la carga del sistema. Cada vez que un proceso cambia de núcleo, se incurre en una sobrecarga de cambio de contexto. Además, al migrar entre núcleos, la caché de CPU que se ha construido para el proceso puede invalidarse, lo que requiere recargar datos en la nueva caché del núcleo al que ha sido asignado.

- **Mejora de la eficiencia de la caché:** Al mantener los procesos en el mismo núcleo, la caché de cada núcleo se mantiene coherente con los datos y operaciones que el proceso está ejecutando, lo que reduce la latencia de acceso a la memoria y mejora el rendimiento general.

En este caso, como los procesos están realizando tareas de cálculo intensivo, asignarlos a un núcleo fijo evita el desperdicio de recursos y optimiza el uso de la caché de la CPU. Esto debería hacer que este enfoque sea más eficiente que dejar que los procesos migren entre núcleos, especialmente cuando cada proceso realiza operaciones repetitivas y de corto plazo (como es el caso de probar contraseñas).

5. Análisis de resultados

He ejecutado cada uno de los scripts en un entorno controlado para medir su rendimiento. Antes de proceder con el análisis cabe destacar la passphrase de es “azz”, esto es determinante para entender los resultados ya que debido a la naturaleza de nuestros scripts, que realizan comprobaciones por orden, se estarían realizando unas 1378 combinaciones antes de llegar a la solución. Debido al bajo número de combinaciones, la diferencia entre algunos de los resultados no es lo suficientemente notable, pero si es suficientemente válida y arroja resultados que nos permiten conocer la eficiencia de los scripts.

Por otro lado, en el caso de los scripts de multithreating y multiprocessing, el trabajo se reparte previamente entre los distintos hilos, por lo que, en este caso, al encontrarse la passphrase al principio de las posibles combinaciones de 3 dígitos, no hay tanta diferencia entre los tiempos en los que se encuentra. Ya que en el caso de 1 hilo, este revisa todas las combinaciones por orden, como se encuentra en la posicion 676, tardaria 676 comprobaciones en encontrar la solucion. En el caso mediante multithreathing y multiprocessing, la carga se reparte previamente por orden entre todos los hilos, por tanto, para 3 dígitos, existen 17576 posibilidades, entre 4 hilos/procesos lanzados, por lo que los 3 ultimos hilos no llegaran a la solucion, si no que sera el primer hilo que tiene que comprobar entre las primeras 4394 posibilidades, el cual llegara a las 676 comprobaciones, al igual que en el caso secuencial. Por lo que, aunque la mejora en el tiempo es notable, puesto que las combinaciones se prueban mas rapido, no es tan notable debido a que la combinacion de 3 digitos sigue tardando lo mismo en probarse. A continuación, presento un resumen de los tiempos de ejecución obtenidos:

Implementación	Tiempo de ejecución (real)
Secuencial (gnupg)	4m28.255s
Secuencial (subprocess)	4m30.498s
Multithreading (4 hilos)	3m53.828s
Multiprocessing (4 procesos)	3m39.659s
Multiprocessing con CPU affinity (4 procesos)	3m04.005s

1. **Ataques secuenciales (gnupg y subprocess)**: Las implementaciones secuenciales, tanto usando gnupg como subprocess, tomaron alrededor de 4m30s en completarse. Esto era de esperarse, ya que ambos enfoques utilizan un único núcleo y no aprovechan la

paralelización. La diferencia entre gnupg y subprocess es mínima, lo que indica que ambos métodos son prácticamente equivalentes en términos de rendimiento.

2. **Multithreading**: El ataque multithreading mostró una mejora en el tiempo real (3m53s). Aunque el tiempo es más bajo que el secuencial, la sobrecarga de la paralelización, el GIL y el tamaño del problema lo hacen ineficiente.

   En este caso (debido al GIL), varios hilos estaban compitiendo por el uso de un solo núcleo, lo que aumentó el tiempo user (14 minutos en lugar de 4 minutos), pero el tiempo real disminuyó un poco debido a la concurrencia limitada.

3. **Multiprocessing**: En contraste, la implementación multiprocessing mejoró significativamente el tiempo de ejecución, reduciendo el tiempo real a 3m39s. Aquí, los procesos independientes se ejecutaron en diferentes núcleos de la CPU, sin las limitaciones del GIL, lo que permitió aprovechar mejor los recursos de la CPU.

4. **Multiprocessing con CPU affinity**: Aunque la versión con CPU affinity también mostró una mejora con respecto a la secuencial, el tiempo real fue ligeramente inferior que el de multiprocessing sin afinidad de CPU, con 3m04s en lugar de 3m39s. En este caso, el hecho de asignar los procesos a núcleos específicos representó una mejora significativa, debido a la naturaleza del problema que consiste en realizar cálculos intensivos de CPU, por lo que evitar los cambios de asignación de cores por parte del sistema operativo mejoró significativamente los tiempos de ejecución.

6. Conclusiones

El multiprocessing fue claramente la opción más eficiente en Python para este tipo de tarea, ya que permitió que los procesos se ejecutaran en paralelo en múltiples núcleos sin la interferencia del GIL. Por otro lado, la opción de multithreading fue más lenta de lo esperado debido al GIL de Python, que bloquea la ejecución concurrente en hilos para tareas intensivas en CPU. Multiprocessing con afinidad de CPU mostró una mejora significativa sobre multiprocessing sin afinidad, lo que sugiere que el sistema operativo estaba realizando numerosos cambios de contexto.

En resumen, la opción más rápida y eficiente fue multiprocessing con affinity CPU y es la que va a ser utilizada para la resolución del problema planteado (passphrase.txt).