Trabajo Práctico Final
Protocolos de Comunicación

Integrantes

- Valentina Novillo - 63156
- Máximo Chiatellino - 63477
- Gonzalo Sharif Curi Martinez - 63463
- Matias Coleur - 63461

Fecha de entrega: 12/02/25

Índice

Protocolos y Aplicaciones

Socks5

El protocolo desarrollado es el proxy Socks5 basado en el [RFC 1928], con su respectiva autenticación descrita en el [RFC 1929]. A su vez se adoptó una política de roles y permisos dentro del proxy, que permiten verificar conexiones a ciertas ips y filtrar a partir de los mismos.
El Socks5 fue implementado utilizando la librería de máquina de estados (stm.c) y el selector (selector.c). Los estados diseñados para el buen funcionamiento del protocolo son:

- **HELLO**:
    - Es el estado inicial donde se hace la conexión con el proxy por parte del cliente.
- **AUTH:**
    - El cliente envía las credenciales de acceso mediante User/Password y el protocolo responde si está autenticado o rechazado.
    - El cliente envía: Versión de la negociación, largo del username, username, largo de la password, password.
    - El proxy devuelve: Versión de la negociación, Status.
    - Los usuarios, sus contraseñas y sus respectivos roles están registrados en un .csv. Se decidió implementar un sistema de Hashing en las contraseñas mediante SHA-3, utilizando la librería openssl. Esta decisión fue producto de que nos resultó poco profesional dejar las contraseñas como texto plano y esta fue una manera de solucionarlo.
- **REQUEST**:
    - En este estado, se procesa la solicitud de conexión enviada por el cliente una vez autenticado. Se leen y parsean los campos definidos en la Sección 4 del [RFC 1928]: Versión, Comando, RSV, Tipo de dirección, dirección destino y puerto destino.
    - Los tipos de dirección soportados son: IPV4, Domain, FQDN, IPV6
    - Se verifica que el comando solicitado sea soportado y que la versión sea la correcta.
    - Se interpreta el tipo de dirección. Si es un nombre de dominio , se debe realizar la resolución DNS, la misma se realiza en un thread aparte para no bloquear el servidor.
    - Se consulta la dirección IP destino y el rol del usuario autenticado. Si la política deniega el acceso, se prohíbe la conexión, responde y termina la comunicación.

- **CONNECT:**
  - Este estado se encarga de establecer la conexión TCP con el destino solicitado.
  - Se crea un nuevo socket para la conexión saliente y se configura en modo no bloqueante.
  - La máquina de estados queda a la espera. Cuando el selector notifica que el socket está listo para escribir, significa que el Three-Way Handshake de TCP termina.
- **REPLY:**
  - Se comunica al cliente el resultado del intento de conexión al servidor remoto.
  - El proxy responde: Versión, respuesta/error, RSV, tipo de dirección.
- **RELAY:**
  - El proxy es intermediario entre el cliente y el socket remoto.
  - Ante la llegada de una escritura en alguno de los sockets, lo lee y lo escribe en el otro receptor.
  - El protocolo guarda la cantidad de bytes transferidos para realizar métricas.
- **DONE**
  - Es el estado final de limpieza y cierre de la sesión para el cliente y el socket remoto.

El diseño implementado realiza “logs” mediante la librería logger.c de la siguiente manera:

ACCESS: Se almacenan los accesos al servidor mediante un usuario, en los mismos se guardan: la fecha del acceso, usuario y contraseña, host al que se solicitó conexión y puerto.

LOGS: Se almacenan todas las acciones que merezcan informe realizadas durante el protocolo, tanto errores como respuestas.

ERRORS: Se informan los errores durante la ejecución del protocolo.

Dissectors POP3/HTTP: Para cumplir la consigna de monitoreo y registro de credenciales, el proxy inspecciona tráfico en claro de POP3 y HTTP. En POP3 se detectan los comandos USER/PASS. En HTTP se detecta Basic Auth y formularios application/x-www-form-urlencoded. Las credenciales detectadas se registran en log/credentials.txt..

Parámetros Recibidos

-l <ip>: Dirección IP donde escuchará el Proxy (Default:127.0.0.1).
-p <puerto>: Puerto TCP para conexiones entrantes del Proxy (Default: 1080).
-L <ip>: Dirección IP para el servicio de Management (Default: 127.0.0.1).
-P <puerto>: Puerto TCP para el servicio de Management (Default: 8080).
-u <user:pass>: Registra un usuario y contraseña válidos (se puede repetir hasta 10 veces).

-N: Deshabilita los disectors de contraseñas (logs de seguridad).

API

El enunciado del trabajo práctico exige que el servidor SOCKS5 pueda recolectar métricas de operación y permitir modificar usuarios o parámetros del sistema en tiempo de ejecución, sin reiniciarlo. Para cumplir esos requerimientos, implementamos una API binaria sobre TCP dedicada exclusivamente al plano de control del sistema.

La API se implementa como un servidor TCP independiente que recibe mensajes request/response con un header fijo y un payload opcional. A través de esta interfaz se pueden ejecutar comandos administrativos, sin interferir con el tráfico de datos del proxy.

Los comandos implementados incluyen:

- Consultas de métricas (process_metrics_request() en api_service.c):
  - conexiones concurrentes,
    `./bin/admin_metrics -C`
  - conexiones históricas,
    `./bin/admin_metrics -H`
  - bytes transferidos.
    `./bin/admin_metrics -B`

- Auditoría por usuario (process_user_connections_request() en api_service.c):
  - obtener todas las conexiones realizadas por un usuario en particular.
    `./bin/admin_metrics -U <user>`

- Gestión de usuarios (process_user_mgmt_request de api_service.c):
  - crear usuarios,
    `./bin/admin_user_mgmt -A <user> <role>`
  - cambiar roles (por ejemplo, usuario ↔ admin),
    `./bin/admin_user_mgmt -R <user> <role>`
  - eliminar usuarios.
    `./bin/admin_user_mgmt -D <user>`

Los comandos están definidos de la siguiente manera:

ADMIN_GET_CONCURRENT_CONN = 0x01,
ADMIN_GET_HIST_CONN = 0x02,
ADMIN_GET_BYTES_TRANSFERRED = 0x03,
ADMIN_SET_USER_ROLE = 0x10,
ADMIN_ADD_USER = 0x11,
ADMIN_DELETE_USER = 0x12,
ADMIN_GET_USER_CONNECTIONS = 0x20,
ADMIN_QUIT = 0xFF,

El archivo `users.csv` inicializa el sistema con un usuario administrador (`admin:admin`), que permite operar la API desde el primer momento.

La API cumple los requerimientos del TP, pero además sigue un enfoque coherente con la arquitectura del sistema. El administrador puede acceder a la API a través del propio servidor SOCKS5, aprovechando el mecanismo de autenticación de usuarios ya implementado. Esto evita duplicar lógica, reduce la superficie expuesta al exterior y permite que toda la administración del proxy se realice utilizando el mismo canal seguro y autenticado.

Client

El cliente SOCKS5 es una herramienta de consola diseñada para probar y validar el funcionamiento de tu servidor proxy. Su ciclo de vida se divide en cuatro etapas secuenciales:

1. **Configuración**: Parsea los argumentos de línea de comandos para saber a qué proxy conectarse y cuál es el destino final deseado.
2. **Conexión TCP**: Establece una conexión TCP inicial únicamente contra el servidor Proxy.
3. **Protocolo SOCKS5**:
    - **Handshake:** Negocia la versión y el método de autenticación.
    - **Autenticación:** Envía usuario y contraseña.
    - **Request:** Solicita al proxy que se conecte al Target.
4. **Túnel (Relay)**: Una vez establecido el túnel, envía una petición HTTP simple y muestra la respuesta en pantalla para confirmar que el tráfico fluye correctamente.

Parámetros Recibidos

- -l <ip>: Dirección IP del servidor Proxy SOCKS5 (Default: 127.0.0.1).
- -p <puerto>: Puerto del servidor Proxy SOCKS5 (Default: 1080).
- -t <host>: Target (Destino final) al que quieres llegar (ej: google.com, 8.8.8.8).
- -P <puerto>: Puerto del Target (ej: 80 para Web, 53 para DNS).

- -u <user:pass>: Credenciales para crear un usuario al iniciar el servidor

Problemas durante Diseño e Implementación

Uno de los puntos que mayor dificultad nos generó fue la integración del selector con la implementación no bloqueante del SOCKS5. La RFC 1928/1929 determina el flujo del protocolo (HELLO, AUTH, REQUEST, CONNECT, REPLY y RELAY), por lo que estructuramos nuestra implementación teniendo en cuenta esa secuencia. Para poder representar toda la información decidimos armar un struct que agrupe todo lo relacionado a la conexión (struct socks5_connection). Este struct contiene los file descriptors involucrados, buffers de escritura y lectura, datos de autenticación, estado actual del protocolo, etc. Lo complejo no fue entender la máquina de estados en sí, sino que funcionara correctamente con el selector. Como cada transición depende de eventos distintos (read, write, block, close), tuvimos que dividir la lógica en callbacks más pequeños y controlar con precisión los intereses de cada fd. La etapa de CONNECT fue particularmente confusa, ya que como la resolución DNS no debía bloquear tuvimos que decidir cómo manejarlo. Finalmente, decidimos ejecutarla utilizando un thread aparte y notificar al selector cuando terminara (usando selector_notify_block)

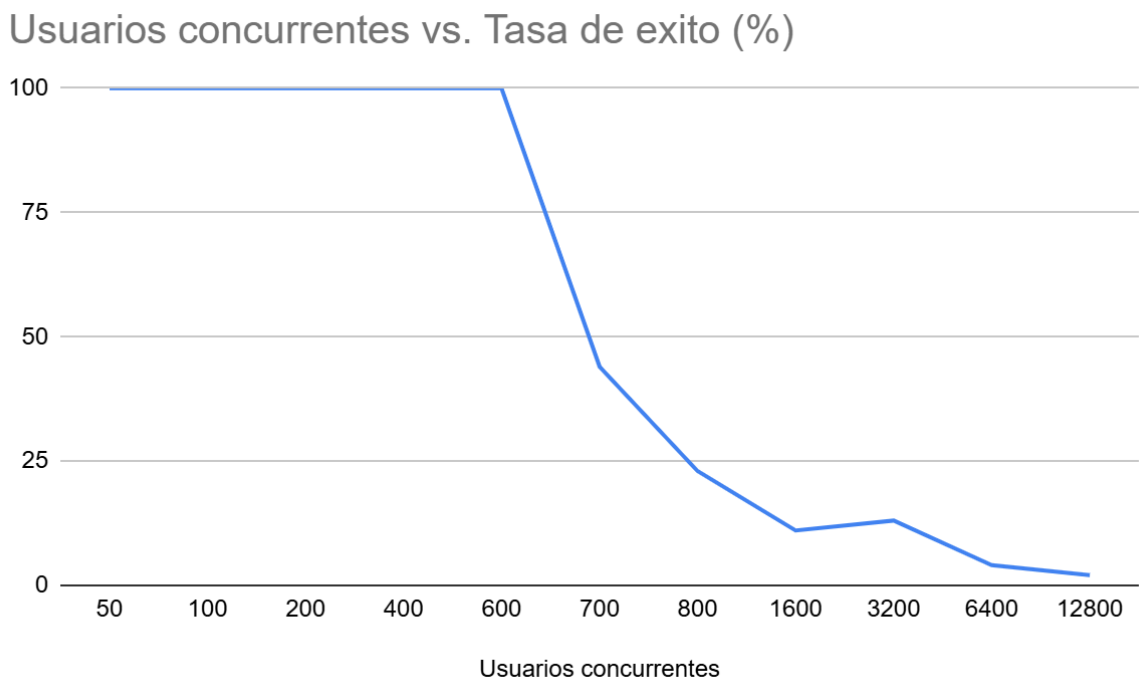
Otro de los problemas de implementación fue el correcto cierre y manejo de los sockets. Si los mismos no se cerraban correctamente, imposibilitaba la reutilización de los files descriptors, por lo que se generaba un cuello de botella y, a la larga, la inutilización del sistema. Esto requirió un análisis preciso de cada instancia de creación de sockets para su correspondiente cierre de manera correcta.

También tuvimos dudas sobre cómo almacenar la información de métricas y logs, ya que no estaba especificado en la consigna. Por lo tanto, terminamos decidiendo escribir todo en distintos archivos .txt.

Limitaciones de la aplicación

Cantidad de usuarios concurrentes

El selector se configuró con MAX_SOCKETS=1024, lo que sugiere un límite teórico de ~512 conexiones simultáneas, considerando que cada cliente consume 2 sockets. Sin embargo, las pruebas mostraron 100% de éxito hasta N=600, por lo que el límite operativo observado es 600 y el punto de quiebre aparece en N=700. En la tabla podemos ver una caída abrupta a partir de N=700 (44% de éxito) y una caída sostenida para Ns mayores, lo que confirma que el sistema entra en saturación y pierde confiabilidad por encima de ese umbral.



Una posible explicación a la diferencia entre el límite teórico (~512) y el punto de quiebre observado (~600) es que el límite del selector aplica a file descriptors simultáneos, pero en el stress test no todas las conexiones mantienen dos sockets activos al mismo tiempo: parte de las conexiones falla temprano, otras cierran rápido, y el scheduling de hilos del cliente escalona los intentos.
Por otro lado, la caída posterior a N=600 se entiende como saturación de recursos: el selector está limitado a una cierta cantidad de sockets (1024), el backlog de listen() es de 128, y el cliente impone un timeout de 5 s; con mayor carga, más conexiones quedan en cola o no completan el handshake a tiempo, por lo que aumenta el número de fallos

Prueba de throughput

Se realizó una prueba de throughput en entorno local, utilizando un echo server como destino (socat TCP-LISTEN:9090,reuseaddr,fork SYSTEM:'cat'), con el objetivo de medir la capacidad de transferencia del proxy bajo carga concurrente. Los parámetros fueron 100 hilos, 10 s de duración y payloads de 20 KB.

Resultados

- Equipo A:
  Sent 2346.00 MiB | Recv 2344.04 MiB | Throughput total 468.71 MiB/s
- Equipo B:
  Sent 15.62 MiB | Recv 13.67 MiB | Throughput total 2.92 MiB/s

Conclusión:
Aunque los parámetros, el comando y el BUFFER_SIZE fueron iguales, el rendimiento

difiere en varios órdenes de magnitud. Esto indica que el throughput está dominado por factores del entorno (logging por paquete, disco/antivirus/OneDrive, uso de CPU o plan de energía) más que por la configuración del test en sí.

Posibles Extensiones

Dentro de las posibles extensiones que se nos ocurren luego de haber trabajado con la implementación actual del servidor, podríamos considerar:

- Implementar otros comandos del protocolo SOCKS5 (**BIND** y **UDP** ASSOCIATE):
  Actualmente solo soporta **CONNECT**. Agregar estos comandos permitiría manejar conexiones entrantes desde el remoto (**BIND**) y tráfico **UDP** (UDP ASSOCIATE), reutilizando la estructura de estados y el registro dinámico de file descriptors.
- Ampliar los métodos de autenticación:
  Hoy solo soportamos autenticación Username/Password pero podrían implementarse otros métodos que también se definen en el RFC de SOCKS5 (como "No Authentication") o incluso métodos adicionales basados en tokens. Esto podría implementarse aprovechando la etapa de **HELLO** ya implementada, donde se negocia el método de autenticación del cliente sin modificar la estructura principal del servidor.

Conclusiones

Trabajar en este proyecto nos permitió desarrollar un servidor SOCKS5 completo, no bloqueante y basado en una máquina de estados, integrando correctamente la librería selector provista por la cátedra. A lo largo de la implementación fuimos tomando decisiones de diseño orientadas a mantener el servidor simple, funcional y estable, como el uso de un hilo separado para la resolución DNS, la incorporación de roles y autenticación con hashing, y la definición de un struct socks5_connection para centralizar toda la información de cada sesión.

Si bien surgieron complicaciones con la gestión de eventos, el cierre correcto de sockets y la representación de métricas, el sistema final cumple con los requisitos del protocolo y permite manejar múltiples conexiones simultáneas de manera eficiente.

A pesar de las limitaciones previamente mencionadas, la implementación deja espacio para futuras extensiones, como la implementación de otros comandos del protocolo o diferentes métodos de autenticación.

En conjunto, el proyecto nos permitió entender en profundidad y trabajar sobre el funcionamiento interno de un proxy SOCKS5, junto con la implementación de una correcta coordinación entre diferentes estados, buffers y file descriptors en un servidor no bloqueante.

Ejemplos de prueba

Se realizaron pruebas funcionales y de stress utilizando los clientes incluidos en el repositorio.

- **Pruebas funcionales**:
  Handshake SOCKS5, autenticación y comandos CONNECT hacia IPv4, IPv6 y FQDN usando `client_ipv4`, `client_ipv6` y `client_dns`.
- **Pruebas de dissectors:**

  Disparo de captura de credenciales en HTTP (Basic Auth o form) y POP3 (USER/PASS) usando client_http_probe y client_pop3_probe, verificando el registro en log/credentials.txt.

- **Pruebas de administración y métricas**:
  Consulta de métricas y modificación de usuarios en tiempo de ejecución mediante `admin_metrics` y `admin_user_mgmt`, siempre a través del proxy SOCKS5.

- **Pruebas de carga y performance**:

  - Concurrencia: `stress_concurrencies`

  - Throughput: `stress_throughput` con servidor de eco local.

Todos los comandos utilizados y ejemplos concretos se encuentran documentados en el archivo **README.md**.

Guía de instalación

La instalación del sistema es directa y no requiere instaladores adicionales.

1. Clonar el repositorio.
2. Verificar dependencias: `gcc`, `make`, `openssl`.
3. Compilar todo el proyecto ejecutando: make clean all
4. Verificar la presencia del archivo obligatorio `users.csv` en la raíz del proyecto.

Los binarios generados se encuentran en el directorio `bin/`.

Instrucciones para la configuración

La configuración del sistema se realiza exclusivamente mediante **parámetros de línea de comandos**, sin necesidad de archivos de configuración adicionales.

- **Admin API**: dirección y puerto configurables con `-l` y `-p`.
- **Servidor SOCKS5**:
    - dirección y puerto del proxy (`-l`, `-p`),
    - endpoint de la API (`-L`, `-P`),
    - usuarios iniciales (`-u user:pass`).

Todas las opciones disponibles y sus valores por defecto pueden consultarse con `-h` en cada binario.

Ejemplos de configuración y monitoreo

Ejemplos representativos del uso del sistema:

Configuración básica:

```
./bin/api
./bin/socks5
```

Configuración explícita:

```
./bin/api -l ::1 -p 8080
./bin/socks5 -l ::1 -p 1080 -L ::1 -P 8080
```

Monitoreo en tiempo de ejecución:

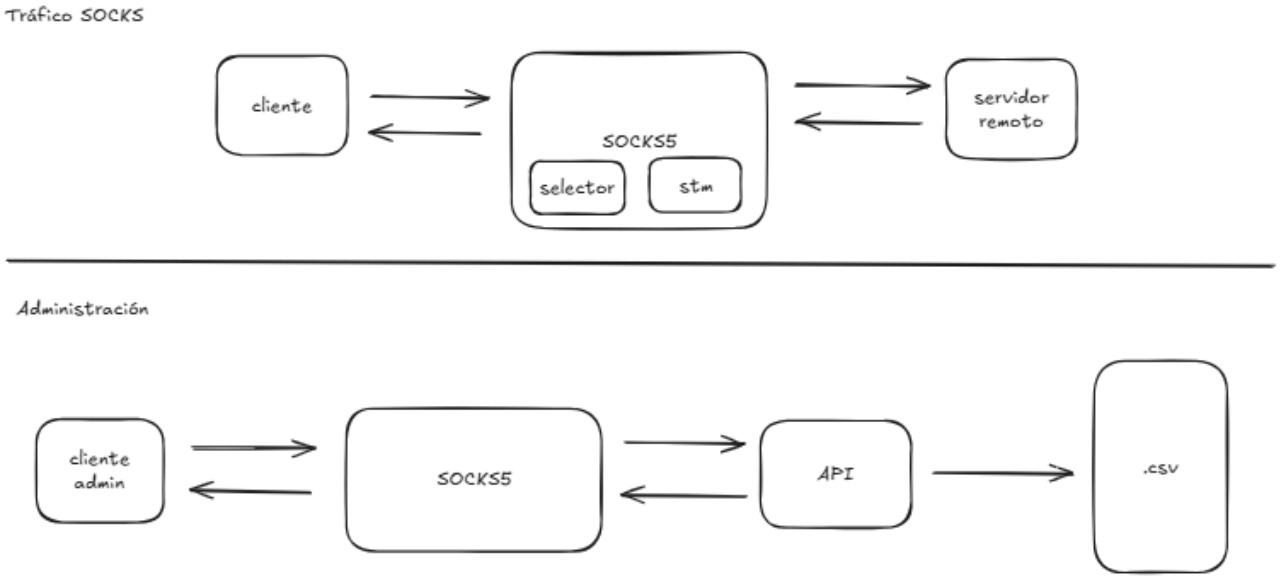
```
./bin/admin_metrics -H -C -B
```

Administración dinámica de usuarios:

```
./bin/admin_user_mgmt -A pepito 1234 user
./bin/admin_user_mgmt -R juan admin
```

La documentación completa de comandos, opciones y ejemplos se encuentra en el archivo **README.md** del proyecto.

Documento de diseño del proyecto



Correcciones

Se realizaron varias conexiones luego de la devolución recibida, las mismas son:

API Configurable

Se realizó la configuración de la IP y del puerto de la API, tanto al ejecutar el socks5 y los clientes que administran y obtienen las métricas.

API No Bloqueante

Se implementó la librería de selector dentro de la API para que la misma pueda soportar conexiones simultáneas no bloqueantes. De esta manera no se genera un cuello de botella dentro de la aplicación y se encuentra en sintonía con lo pedido.

Protocolo de conexión entre la API y el Socks5

El protocolo de comunicación entre la API y el Servidor Socks5 es un protocolo binario personalizado diseñado sobre TCP. Sigue un esquema de Header Fijo + Payload Variable, lo que permite un parseo eficiente y seguro.

Cada mensaje comienza con un Header de 4 bytes, seguido opcionalmente por un cuerpo de datos:
| VER | CMD | LEN | PAYLOAD |

- **VER (1 Byte):** Versión del protocolo (actualmente 0x01). Permite evolución futura sin romper compatibilidad.
- **CMD (1 Byte):** El código de operación. Define la acción a realizar o el estado de la respuesta.
    - Ejemplos Request: 0x01 (Get Metrics), 0x05 (Add User).
    - Ejemplos Response: 0x00 (Success), 0x01 (Error).
- **LEN (2 Bytes):** Longitud del Payload en bytes.
    - **Importante:** Se transmite en Network Byte Order (Big Endian).
    - Debe ser convertido usando htons() al enviar y ntohs() al recibir.
    - Si LEN = 0, el mensaje no tiene cuerpo.
- **PAYLOAD (Variable):** Los datos específicos del comando (ej: nombre de usuario, contraseña, o el valor de una métrica). Su tamaño exacto está definido por el campo LEN.

Conexión a varios dominios

Se solucionó un error que impedía establecer conexión con dominios que resuelven a múltiples IPs. Anteriormente, un fallo en la primera dirección IP provocaba la caída del servidor Socks5. Se ha corregido la lógica de reintento que recorre todas las direcciones devueltas por el DNS hasta lograr una conexión exitosa o agotar la lista.

Tamaño del buffer

Durante el coloquio se observó que la performance obtenida no fue óptima.

Una posible causa identificada fue el uso de un tamaño de buffer inadecuado en las operaciones de lectura y escritura.

Como corrección, se evaluó el impacto del tamaño del buffer en la cantidad de llamadas send() y recv(), observándose que buffers más grandes permiten transferir más datos por llamada, reduciendo el número total de syscalls y, con ello, el overhead del kernel.

En consecuencia, se incrementó BUFFER_SIZE de 4096 a 16384. Este cambio redujo la cantidad de llamadas send/recv para el mismo volumen de datos y mejoró el rendimiento general del relay. En nuestras pruebas, el throughput se duplicó aproximadamente, resultado coherente con esa reducción de overhead.

Write optimista

Se observó que el ciclo de select utilizado originalmente resultaba ineficiente, ya que delegaba innecesariamente en el selector la notificación de disponibilidad de escritura aun cuando el socket podía aceptar datos inmediatamente.

Como mejora, se optimizó el flujo de I/O implementando el esquema:

select → read → write

De esta forma, luego de una lectura exitosa se intenta escribir directamente en el socket destino.
Solo en el caso de que la escritura no pueda completarse (por ejemplo, debido a un EWOULDBLOCK/EAGAIN en sockets no bloqueantes), se vuelve a delegar en el selector la espera de disponibilidad de escritura.

Esta optimización reduce cambios de estado en el selector, disminuye la cantidad de iteraciones del ciclo de eventos y mejora la eficiencia general del relay.

Password como input

Atendiendo a las correcciones de seguridad señaladas, hemos modificado el mecanismo de ingreso de credenciales en el cliente. Anteriormente, la contraseña se pasaba como argumento de línea de comandos, lo que la exponía en el historial de la shell y en la lista de procesos del sistema. Ahora, la contraseña se solicita de manera interactiva a través de la entrada estándar, asegurando que no quede registro visible ni persistente una vez finalizada la ejecución.

SO_REUSEADDR y Carpeta “logs”

Se configuró la opción en el socket del servidor Socks5 (Ya existente en la API). Esto permite reiniciar el servidor inmediatamente sin esperar a que el sistema operativo libere el puerto, evitando el error "Address already in use".
A su vez, se agregó una validación al inicio del programa que verifica la existencia del directorio logs/. Si no existe, la aplicación lo crea automáticamente, así como los propios archivos, previniendo fallos de escritura o cierres inesperados por falta de la ruta de destino.