Trabajo Práctico Especial
Protocolos de comunicación
Segundo cuatrimestre
2020

- Integrantes: Brandy, Tobias
  Pannunzio, Faustino
  Sagüés, Ignacio

- Fecha de primer entrega: 17/11/2020

1.Aplicación desarrollada

Para el siguiente trabajo práctico se construyó un servidor proxy socks que sigue la versión 5 de dicho protocolo explicada en el RFC 1928. El mismo, permite autenticación con el cliente mediante usuario y contraseña, como se explica en el RFC 1929. De no proveerse esta opción de autenticación, el servidor permitirá acceder sin autenticación, asignándole a la nueva sesión un usuario anónimo (user:anonymous, pass: anonymous).

El proxy soporta consultas de conexión que podrán ser resueltas tanto recibiendo la IPv4, IPv6 o el nombre del dominio del servidor al que se quiera conectar. Los nombres de dominio son resueltos mediante DNS sobre HTTP obteniendo los valores tanto de los registros A, como los AAAA. Para la construcción de esta última funcionalidad se siguieron los lineamientos planteados en el RFC-8484.

A su vez, se debió implementar un protocolo propio, detallando su funcionalidad e implementación en el documento `pipoProtocol.txt`. Este protocolo, funciona sobre SCTP, es binario y está orientado a sesión. Sobre él se tuvo que implementar un flujo de administración que le permita a un cliente, luego de autenticarse como administrador del servidor, pedir información sobre el estado del proxy, métricas y parámetros del mismo. A su vez, se permite modificar valores pertinentes del servidor como timeouts, memoria utilizada en I/O, y administración de usuarios.

Para poder probarlo correctamente, se construyó un cliente interactivo que mediante SCTP, se conecta con el servidor y recibe de entrada estándar el nombre de usuario y contraseña necesarios para obtener una terminal de administrador. Una vez autenticado, se recibirá por línea de comandos la funcionalidad deseada, enviará el pedido al servidor, y se encargará de imprimir a salida estándar la respuesta dada por este último.

Como funcionalidades adicionales del servidor se implementaron:

- Almacenamiento volátil de métricas del servidor (detalladas en `pipoProtocol.txt`)
- Generación de logs por salida estándar de accesos por usuario siguiendo los lineamientos del manual del servidor.
- Obtención de usuarios y contraseñas que pasen por el servidor a través de los protocolos HTTP y POP3, junto con la apropiada generación de logs por salida estándar siguiendo los lineamientos del manual del servidor.
- La capacidad de agregar y borrar usuarios durante la ejecución del servidor, tanto administradores como usuarios estándar.
- Una rutina de limpieza de sesiones inactivas (por default 15 segundos de inactividad, pero se puede configurar al inicializar o en ejecución, y hasta deshabilitar la funcionalidad).

Estructura Principal del Servidor

Como principal mecanismo para atender múltiples conexiones, el servidor implementa multiplexación de sockets basado en la syscall pselect.

Todo socket activo en el servidor se encuentra registrado en una estructura de datos que encapsula dicho pselect (Selector). Los sockets cliente, servidor, DNS, administradores y pasivos se encuentran registrados. Dado que pselect tiene un límite de 1024 sockets, el servidor no podrá soportar más de 510 conexiones cliente establecidas por IP (menos aún en el caso de DNS).

Toda conexión tiene asociada una estructura Sesión donde se almacena todo el estado e información de la misma. Los estados y transiciones de una sesión se encuentran parcialmente documentados en `src/states/socksStateMachine.dot`.

2.Problemas encontrados durante el diseño y la implementación

Sesiones que no cerraban la conexión

Las primeras veces que conectamos el proxy al browser, nos dimos cuenta que este utilizaba mucho la estrategia de no cerrar conexiones, a pesar de ya haber terminado la transacción. En algunos casos, la cantidad de conexiones activas podía llegar a sobrepasar las capacidades de nuestro server. Es por eso que decidimos implementar una rutina de limpieza, que cierre y libere todas las sesiones que no hayan tenido actividad en los últimos 15 segundos. Así, las chances de que nuestro servidor llegase a su límite, bajan.

De todas maneras, decidimos hacer esta funcionalidad opcional, y también permitir configurar el timeout máximo que una sesión puede estar inactiva.

Resolución de nombres

A la hora de establecer una conexión con el comando connect existe la posibilidad de indicar el destino de conexión utilizando una dirección IP o un domain name. El caso de la IP consiste en pasear de manera correcta la dirección provista por el usuario, iniciar un intento de conexión y notificar al usuario el resultado del mismo. Por otro lado, el caso de un domain name conlleva un procedimiento más complejo. Si bien este proceso podría realizarse utilizando el protocolo DNS, como parte del enunciado se solicita resolver estas solicitudes vía DoH. Teniendo en cuenta que este protocolo está diseñado utilizando DNS sobre HTTPS, y este último

no forma parte de los contenidos de la materia ni tenemos el conocimiento suficiente para implementarlo, la alternativa utilizada fue enviar solicitudes DNS sobre HTTP.

Método HTTP a utilizar

Una de las decisiones a tomar a la hora de implementar un cliente DoH es la elección del método HTTP a utilizar. El RFC 8484 define como válido tanto al método GET como al método POST. Teniendo en cuenta que ambas son válidas y soportadas por los servidores, la elección de un método a utilizar se basó en nuestros criterios. Habiendo tenido esa libertad, optamos por utilizar como método default GET. Esto se debe a su carácter idempotente y cacheable. Teniendo en cuenta el costo extra que agrega establecer una nueva conexión y conseguir el resultado del servidor DoH, la posibilidad de cachear las respuestas de forma local permitiría reducir el tiempo de este proceso y acercar el tiempo de respuesta al de una solicitud vía IP. Además, teniendo en cuenta que la longitud de un domain name está limitada y los mensajes DNS, dado su carácter binario, son muy pequeños, concluimos que no habría problemas de espacio al utilizar una query via GET. De todas formas, dada la similitud entre implementar ambas opciones, se optó por ofrecer la opción de utilizar el método POST como un ajuste de configuración más vinculado a DoH.

Proceso de resolución

Este proceso consiste en generar solicitudes DNS basadas en la estructura descrita en el RFC 1035. Una vez generados los paquetes DNS es necesario codificarlos con base 64 para poder utilizar el resultado como query param de una solicitud HTTP. A la hora de obtener los resultados enviados en la respuesta es necesario parsear el mensaje HTTP en busca, primero de un status que indique que el pedido fue recibido correctamente y luego, obtener la respuesta DNS binaria del body del mensaje. Para simplificar el proceso de parseo de la respuesta, se decidió realizar la consulta en HTTP 1.0. Esto tiene la contra de que no podemos aprovecharnos de la funcionalidad de conexiones persistentes de la versión 1.1.

Robustez

Para brindar un servicio robusto, el servidor resuelve el dominio tanto para IPv4 como para IPv6 y solo se considera que no se pudo establecer conexión en caso de que ninguna de las IPs recibidas nos permita lograrlo. Para poder realizarlo se tuvieron en cuenta dos alternativas, realizar los pedidos en serie o en paralelo.

Por un lado, realizar el proceso en serie reduce la complejidad del manejo de estados. Consiste en realizar el proceso de generación, envío y parseado de query y el intento de conexión una vez y, en caso de error (fallo en la conexión, o ninguna de las IP obtenidas eran válidas), realizar todo de nuevo. Por otro lado, la principal ventaja de realizar las solicitudes en paralelo es reducir el tiempo requerido para

este proceso. Teniendo en cuenta estas dos opciones, decidimos priorizar el paralelismo en pos de un mejor tiempo de respuesta.

Conexiones al servidor

Para poder lograr estos objetivos se decidió que cada sesión tiene control de la conexión al servidor DoH. Se genera un par de sockets particulares que solo serán utilizados por esa sesión. Esto nos permite que cada cliente tenga independencia por sobre las solicitudes de los demás. Por otro lado, utilizar una conexión por cada solicitud nos permite evitar la complejidad que trae centralizar las solicitudes.

Por supuesto, esta estrategia tiene una clara desventaja que no se puede no mencionar. La cantidad de conexiones simultáneas que podemos tener con el servidor DNS pueden llegar a ser muy altas, lo que significa un uso bastante elevado de recursos que podría ser evitado. Además, considerando que nuestro servidor tiene un límite duro de 1024 conexiones, la cantidad de clientes simultáneos que realizan pedidos DNS no puede superar los 340.

Consideración

Teniendo en cuenta que el servidor utiliza un servicio de limpieza de conexión inactivas, es importante que todos los intentos de conexión sean realizados. Si una de las direcciones IP obtenidas demora mucho en establecer conexión o declarar estado de error, la sesión podría ser limpiada por inactividad. Dejando como resultado una lista de IPs posibles sin probar.

Para poder lograr esto fue necesario limitar el tiempo que un socket puede estar inactivo esperando a establecer conexión. Para lograrlo se limitó la cantidad de paquetes SYN que son reenviados al no recibir respuesta mediante la opción TCP_SYNCNT. Si bien esta opción parece no ser compatible con todos los Sistemas Operativos (solo es modificable en versiones de Linux superiores a 2.4), consideramos que el beneficio de tener control sobre el tiempo máximo que puede tomar un intento de conexión valía la pena.

Sockets pasivos

El servidor deberá poder recibir durante todo el tiempo que esté activo (siempre y cuando haya lugar disponible) conexiones de distintos clientes tanto por IPv4 como por IPv6. Para poder realizar esto, se levanta un socket pasivo para el primer tipo de dirección y otro para el segundo, ambos esperando a recibir nuevas conexiones. A su vez, en base al tipo de dirección que se reciba para el servicio de administración, se levanta el socket del tipo especificado, IPv4 o IPv6 (por defecto 127.0.0.1). Estos tres sockets están durante todo el proceso de funcionamiento del servidor dentro del Selector, por lo cual se limita al máximo de entradas libres en este a 1021 sockets. Cabe aclarar que ninguno de estos sockets resulta bloqueante.

Manejo de estados

Para la construcción de la máquina de estados principal, se decidió separar a los estados en distintas clases. A cada uno, se le asignó un handler específico encargado de ejecutarse cada vez que el socket sea elegido por el Selector, un handler para lectura, y otro para escritura. Esto fue realizado de esta manera para evitar acoplamiento entre las funcionalidades de estados y para poder simplificar el entendimiento del código, dado que cada uno tiene únicamente una funcionalidad. Además de los handlers previamente mencionados, a todos los estados se le asignó una función que se ejecuta luego del cambio de estado. Esta se encarga de inicializar los parámetros específicos del nuevo estado al que se ha entrado, y se ocupa de establecer los nuevos intereses del selector en base al tipo de estado que se encuentre, para de esta manera no ser llamado por el selector de manera incorrecta y evitar el busy waiting.

Manejo de memoria

El manejo de memoria es uno de los problemas de mayor importancia que nos encontramos durante el proyecto. Cuando se necesita almacenar información hay dos alternativas, utilizar memoria estática o dinámica. Por un lado, la utilización de memoria estática es mucho más segura, toda el espacio es reservado en el stack durante la ejecución y liberado cuando la función que lo genera termina, pero requiere reservar cotas máximas, que muchas veces se encuentra muy por arriba del caso promedio, algo que lleva a desperdiciar mucha memoria. Por otro lado, la memoria dinámica es mucho más práctica, se puede reservar en momento de ejecución en base a las necesidades de ese momento, con las contras de que puede fallar, y que hay encargarse de liberarla cuando ya no se necesite. Teniendo en cuenta que no contamos con control directo sobre cuándo se reservan y liberan recursos (debemos liberar toda la memoria ante una señal de cierre, que puede suceder en casi cualquier momento), buscamos alcanzar el mejor balance posible entre ambas.

Casos de usos para memoria dinámica

Primero y principal, para almacenar la estructura de control principal de la sesión se utiliza memoria dinámica, esto nos permite reservar el espacio necesario a medida que se establecen nuevas conexiones. Además, permite reducir los requisitos de memoria iniciales a la hora de utilizar el servidor.

Sumado al manejo de sesión, los buffers utilizados para el intercambio y procesado de información utilizan memoria dinámica. Las dos principales razones son, la posibilidad de modificar su tamaño en tiempo de ejecución y generar una aplicación que pueda ser adaptada a las condiciones de ejecución. Por otro lado y, sumando a lo detallado respecto a la sesión, si todas las sesiones y buffers de todas las posibles conexiones del servidor se utilizaran en memoria estática, las

cantidades mínimas de memoria necesaria para la ejecución sería innecesariamente alta.

Por último, los otros lugares que se ven beneficiados por la memoria dinámica son las listas de direcciones IP obtenidas al parsear las respuestas DNS, el mapa utilizado para almacenar los usuarios logeados (mismas consideraciones que el manejo de sesiones) y el encabezado que se ocupa de guardar la información pertinente al DNS, pues la cantidad de memoria requerida por este encabezado no es menor, y las conexiones que se realizarán por IP no lo necesitan.

Liberación de memoria

Nuestra estrategia para la liberación de memoria fue que dependa casi exclusivamente de la función de liberación asociada al Selector. Así, de haber una señal de finalización, lo único que debemos hacer es finalizar el Selector, y este se encargará de llamar a las respectivas funciones de liberación. Esta es una de las principales razones por la cual cada caso de alocación de memoria debía ser profundamente analizado.

Connect no bloqueante

Como fue explicado previamente, se decidió construir una máquina de estados en la que cada uno de ellos tiene un handler que le indica cómo proceder al momento de ser levantado por el selector. De este modo, la sesión se levanta solo cuando ha ocurrido un evento que ha causado que el socket sea seleccionado. El problema puntual surge a la hora de utilizar la directiva connect, ya que si no se desea que el servidor quede bloqueado, se tendrá que establecer la conexión en dos pasos. El primero, para indicar la intención de conectarse y, en caso de obtener como errno EINPROGRES, registrar el socket a la espera de un evento de lectura. Una vez que se llame nuevamente a la sesión, en un estado diferente, se pregunta por las opciones del socket con el que se intenta establecer conexión, y asegurarse de que no haya ocurrido un error. Este procedimiento se ejecuta de esta forma cada vez que se usa connect, tanto cuando se elige la opción de conexión por IP como cuando se recibe un nombre de dominio. No solo eso, sino que al establecer la conexión con el servidor dns, estos dos pasos deben ser ejecutados. Esto si bien puede resultar tedioso, nos asegura que la conexión se establece correctamente y que no se está bloqueando el servidor, por lo que se consideró que era el camino a seguir.

Tamaño de los buffers entrada salida

Teniendo en cuenta que nuestra aplicación es el intermediario de una comunicación y que la información está constantemente entrando y saliendo, es importante que los buffers utilizados para mover esta información tengan un tamaño adecuado. Por un lado un buffer muy chico implica que hay que hacer muchas

operaciones para poder mover una cierta cantidad de información. La aplicación estaría constantemente en ejecución ya que siempre tiene información para leer y escribir. Por otro lado, si bien un buffer extremadamente grande garantiza que todo lo que hay por leer tenga donde ser almacenado y siempre haya contenido de sobra para escribir, el espacio utilizado por la aplicación sería extremadamente grande también. A este efecto se le suma que, por cada sesión nueva, se agregan nuevos buffers de este gran tamaño. Es por esto que la elección de un buen tamaño es una decisión a tener en cuenta.

Para determinar esto se realizaron pruebas similares a las de performance que se mostrarán a continuación con distintos tamaños de buffer y analizando los distintos tiempos de transferencia del archivo arribamos a un tamaño de 4Kb.

A pesar de este valor base, gracias al protocolo de administración, este valor puede ser incrementado en tiempo de ejecución para ajustar la performance a la demanda.

Vaciado de los buffers ante un cierre de conexión

A la hora de cerrar una conexión establecida entre un cliente y un servidor, ya sea por parte del primero o del segundo, hay que ocuparse de que se vacíe correctamente el buffer de quien cerró la conexión, a quién llamaremos closer, ya que si bien ya no podrá enviar nueva información, si terminará de enviar la que ya se había escrito.

En el caso del otro extremo de la conexión que aún no ha sido cerrado, a quien llamaremos closy, podrá seguir enviando información normalmente, hasta que decida cerrarse, y de igual forma que el closer, deberá terminar de enviar los bytes que hayan sido escritos hasta ese momento en su buffer de salida. Una vez vaciados los buffers tanto del closer como del closy, se procede a cerrar la conexión.

Para poder ir informando los cierres de conexión tanto al closer como al closy mediante TCP, sin que el sistema operativo nos quite los recursos asociados al socket, utilizamos la syscall shutdown.

Manejo de diferentes estados durante una conexión

Durante todo el proceso en que un usuario se conecta con el proxy, se le brinda una estructura Sesion, con sus datos, los respectivos a la conexión con el servidor, sus Buffers de entrada y salida, estado actual dentro de la máquina de estados, etc. En particular, se brinda un socksHeader, que se encargará de brindar los recursos necesarios para resolver el estado en que se encuentre, siendo estos principalmente los parsers de los datos que está enviando. Distintos estados dentro del proceso, como pueden ser el Hello y el pedido del usuario, tienen encabezados distintos.

Dado que estos estados durante la conexión son disjuntos, y para no desperdiciar memoria, esta estructura es una unión entre distintos headers

específicos para cada momento de la conexión, como pueden ser proceso de Autenticación, de Request, etc.

El único caso que resultó siendo particular, fue el de DNS, dado que si bien es un header particular, está contenido dentro del proceso de Request, por lo que no parece una opción correcta la de sobrescribir dicho encabezado. Como mencionamos anteriormente, dado que se comprende que no todas las sesiones harán uso de este header, se lo aloca de manera dinámica y se lo libera una vez terminado el pedido de DNS.

Logs no bloqueantes

Un requerimiento del servidor es que ninguna operación de entrada/salida puede ser bloqueante. Esto incluye los logs a salida estándar (y error estándar). Para poder cumplir con el requerimiento, pero no perder la inmediatez de los logs, decidimos marcar estos file descriptors como no bloqueantes e intentar hacer la escritura de manera normal. Solo si dicha escritura hubiese sido bloqueada (EWOULDBLOCK), cargamos el log a un buffer y registramos el interés de escritura para dicho fd en el Selector.

Lo dicho previamente no se cumple fuera del loop principal del servidor. En las etapas de inicialización y finalización del servidor, decidimos seguir utilizando funciones de escritura bloqueantes (ej. fprintf), pues queremos asegurarnos de que no haya posibilidad que dichas escrituras no se hagan. Entendemos que esta decisión no es mala, pues fuera del loop principal el servidor no le está ofreciendo servicio a ningún cliente.

Obtención de credenciales

A la hora de implementar la obtención de credenciales de los protocolos HTTP y POP3, fue necesario tomar distintas decisiones acerca de qué funcionalidades de cada protocolo íbamos a soportar.

En primer lugar, decidimos que solo íbamos a tomar en cuenta las credenciales exitosas y no posibles credenciales.

HTTP

Buscamos credenciales dentro del header Authorization que hayan utilizado el esquema Basic. Soportamos tanto 1.1, cómo 1.0. No soportamos conexiones persistentes, esto quiere decir que solo buscamos el header Authorization en el primer mensaje del cliente, y luego dejamos de buscar. Para verificar que las credenciales fueron válidas, nos fijamos que la respuesta del servidor contenga un código 2XX.

POP3

Buscamos los comandos USER y PASS en todas las comunicaciones del cliente. Permitimos que haya otros comandos (inválidos o no) antes y después del comando USER. Una vez enviada la password, verificamos que el servidor responda con un '+'. No soportamos pipelining.

La mayoría de estas decisiones apuntan a reducir lo máximo posible la cantidad de caracteres que haya que analizar de la comunicación, para reducir el impacto en la performance.

Borrado de usuarios

A la hora de implementar el borrado de usuario, nos encontramos con el problema de qué hacer con las sesiones activas que podía tener dicho usuario en ese momento. Nuestra solución fue cerrar y liberar los recursos de todas las sesiones activas asociadas al usuario.

3.Limitaciones de la aplicación

Límite de conexiones simultáneas

Dado que el Selector cuenta con un máximo de 1024 sockets a la vez por su implementación interna de pselect, nos encontramos con una clara limitación del número de conexiones que se pueden establecer.

En el mejor de los casos, una sesión involucra 2 sockets, esto nos deja en un máximo de 512 sesiones. Si a esto le sumamos que debemos tener registrados 3 sockets pasivos y 2 fd de salida estándar, el límite baja a 509 sesiones.

Capacidad de conexiones acotadas en caso de que usen DNS

Dando por sentada la limitación anterior, existe otra relacionada al proceso de resolución de nombres de hosts.

El problema surge a la hora de utilizar DoH, ya que para conectarse con este servidor se abrirá un nuevo socket por cada pedido dns, uno preguntando por IPv4 y otro por IPv6. El lado positivo es que esto ocurre antes de establecerse la conexión con el servidor, y para cuando este último socket se ha abierto, los pertinentes al DNS ya han sido cerrados. Por lo que la mayor cantidad de descriptores que se pueden estar ocupando para una sesión sería de 3.

Con esto último en mente, en un hipotético caso de que todos los clientes se conecten mediante DNS y logren establecer conexión con los dos pedidos a DNS, el máximo de conexiones simultáneas, teniendo en cuenta los sockets pasivos, será de 340, un número mucho menor al previamente mencionado.

Cantidad de queries DoH en paralelo

Por la forma en que se diseñó y se implementó el manejo de solicitudes de DNS, si se quisiese incorporar una o más solicitudes en paralelo habría que hacer un cambio en la lógica de los estados asociados. Todo el proceso de envío, garantizado de recepción, parseo e intento de conexión se implementó de forma poco extensible, siempre asumiendo que son solo dos conexiones paralelas. Por otro lado, si se desean generar pedidos en serie como se mencionó anteriormente, se debería poder hacer con mayor facilidad.

Limitaciones respecto a usuarios

Decidimos que la máxima cantidad de usuarios sea 255. Además, ninguna de sus credenciales puede superar los 255 caracteres de largo. Consideramos que estos tamaños eran sensatos teniendo en cuenta el scope del servidor.

4.Posibles extensiones

Mejoras Selector

Primero y principal, el primer punto mejorable del servidor es su corazón, el Selector. Las mejoras pueden ser múltiples.

Desde lo más básico, se podría cambiar su implementación de lista que asume que los file descriptors se encuentran contiguos, por un hash table. Esto permitirá que se encuentren dispersos, ahorrando recursos, y permitirá acceder a información específica de cada uno en orden 1.

Una mejora más sustancial sería la de cambiar la utilización de la syscall pselect por otra más potente, como poll o epoll. Esto resolvería el límite de 1024 fds que tiene actualmente el servidor, y mejoraría la performance del mismo cuando haya una alta cantidad de fds registrados en el mismo.

Una mejora que fue charlada en clase, pero que requiere una reestructuración del servidor y que se encuentra a un nivel mucho más alto de dificultad es la de implementar el patrón Reactor para aprovechar los recursos multicore, y así poder atender varias sesiones en simultáneo.

Por último, una mejora menor consiste en monitorear más detenidamente el límite de fds manejados por el selector para suscribir y desuscribir los sockets pasivos cuando se alcanza el límite. Esto permitiría evitar rechazar conexiones que no pueden ser procesadas actualmente y dejarlas esperando hasta que sea posible aceptarlas.

Mejoras implementación DoH

Consideramos que nuestra implementación de resolución DNS mediante DoH es muy mejorable.

Idealmente, se podría implementar los pedidos sobre HTTP 1.1 y así poder aprovechar la funcionalidad de conexiones persistentes que ofrece el protocolo. El servidor podría tener una pool de conexiones ya establecidas con el servidor DNS y cuando una sesión necesite hacer un pedido, utilice alguna de estas conexiones.

Con esta implementación, la cantidad de establecimientos de conexión con el servidor DNS bajaría muchísimo, junto con la cantidad de sockets abiertos.

Por otro lado, se podría tener la opción de resolver nombres a través de DNS normal, como fallback.

Completa implementación del protocolo socks5

Primero y principal se podría expandir el proxy para que implemente los otros dos comandos descritos en el protocolo socks5, bind y udp. Además, se podría implementar GSSAPI como posible método de autenticación. Con estas dos incorporaciones el proxy sería capaz de manejar todas las funcionalidades de un proxy socks5.

5.Conclusiones

Consideramos que el diseño utilizado para multiplexar los eventos de entrada y salida sumado a la abstracción de una máquina de estado y el uso de clases con métodos suppliers es una estrategia muy poderosa y extensible. Por un lado se genera un manejo transparente y desacoplado de todos los estados. El encargado de llamarlos nunca necesita saber quiénes son, dónde está y que necesita. Por otro lado, abstrayendo la implementación del multiplexado se pueden realizar cambios a la implementación sin afectar al manejo de estados.

También se analiza que el desafío de mantener dos descriptores trabajando en paralelo, para realizar las preguntas al dns correspondientes a IPv4 e IPv6, teniendo que considerar la integridad de ambos en todo momento para analizar cómo trabajan en conjunto, resultó complejo, pero muy formativo e interesante.

6.Ejemplos de prueba

Para poder testear y verificar el correcto funcionamiento del sistema de resolución de nombres vía DoH se utilizó un Nginx como reverse proxy de un servidor DoH online. El principal objetivo del mismo es convertir la solicitudes locales HTTP en solicitudes HTTPS aceptadas por estos servidores ya que, como fue mencionado previamente, nuestro servidor no maneja este protocolo. La configuración del mismo fue la siguiente:

```
server {
    listen  8053;
    server_name doh;

    location / {
      resolver 1.1.1.1 ipv6=off valid=30s;
      set $empty "";
      proxy_pass https://doh.opendns.com$empty;
    }
 }
```

En el ejemplo se puede ver la URL del servidor DoH provisto por Cisco aunque este no fue el único servidor utilizado. A lo largo del desarrollo y testeo se utilizaron otros servidores como Google, Quad9, Cloudflare.

Dado que para algunos casos el ambiente de desarrollo utilizado no contaba con conexión IPv6 fue necesario forzar a nginx a resolver y conectarse únicamente vía IPv4. Teniendo en cuenta esto, las pruebas de conexiones vía IPv6 fueron realizadas principalmente a un servidor Nginx local distribuyendo un archivo estático o comportándose como un reverse proxy de alguna página web.

Además de la utilización de nginx para poder ejecutar las queries, la herramienta strace nos permitió comprobar el correcto flujo de las dos conexiones y el proceso de intentar con distintas direcciones hasta conseguir una que conecte. Anexamos un breve extracto de strace comentado donde se puede ver el flujo de un intento de conexión fallido seguido de uno exitoso (ver anexo).

Loggeo no bloqueante

Teniendo en cuenta que una de los requisitos principales del TP es trabajar con entrada y salida no bloqueante, resultó de gran importancia testear correctamente que los logs se imprimen de manera no bloqueante. Para esto diseñamos un breve programa al cual se le redirige la salida estándar de nuestro servidor. Su funcionalidad consiste en generar esperas de 10 segundos en las cuales no hay nadie escuchando y, por ende, la escritura a stdout sería bloqueante.

```
int main(int argc, char const *argv[]) {

    while(1){

        sleep(10);

        for(int i = 0; i < 1000; i++)
            putchar(getchar());
    }
```

```
    return 0;
}
```

Otras pruebas generales

Para el análisis estático del código utilizamos cppcheck y pvs studio, con los cuales se encontraron algunos errores menores y gracias a eso se pudo detectar una fuga de memoria y un caso no contemplado. Los problemas puntualizados por cppcheck que no fueron solucionados consisten principalmente en reducir el scope de ciertas variables que solo se usaban dentro de un ciclo, pero se decidió descartar la sugerencia para priorizar el estilo del código. Otro de los falsos positivos encontrados es que se analiza un leak de memoria en request.c en el caso de error. Sin embargo, esos recursos son liberados en el estado final al que se llega luego de enviar el mensaje de error.

También se usó scan-build y el nivel 3 de optimización de Clang y GCC, con el que se encontraron algunas variables que no estaban siendo inicializadas y otras problemas con tamaños de datos incorrectos.

Durante el desarrollo, siempre se incluyó el flag -fsanitize=address para ayudarnos a detectar usos incorrectos de la memoria.

Performance

Para analizar el nivel de pérdida de performance que generaba pasar por el proxy en relación a no hacerlo se realizó una sencilla prueba, pidiéndole un archivo estático local de gran tamaño a un nginx mediante curl guardándolo en un directorio local. Se trabajó con dos archivos, uno de 1.5 GB y otro de 3GB. Para cada uno de ellos se realizaron dos pedidos, el primero sin pasar por el proxy y el segundo pasando por este. A su vez, estos pedidos se realizaron en distintas computadoras, dado que lo que se buscaba analizar es la diferencia entre casos y no la velocidad de cada uno, dato que puede estar relacionado con factores externos.

Archivo de 1.5 GB:

```
faus@LAPTOP-1020R6JC:/tmp$ curl localhost/oneHalfGig > /tmp/gig
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1536M  100 1536M    0     0   884M      0  0:00:01  0:00:01 --:--:--  884M
```

Caso sin pasar por proxy

```
faus@LAPTOP-1020R6JC:/tmp$ curl localhost/oneHalfGig -x socks5://localhost:1080 > /tmp/gig
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 1536M  100 1536M    0     0   554M      0  0:00:02  0:00:02 --:--:--  554M
```

Caso pasando por el proxy

Como puede comprobarse y era esperado, la performance al pasar por el proxy disminuye. La diferencia de tiempo entre casos es de aproximadamente la mitad, lo cual tiene sentido dado que el proceso de establecer la conexión con el proxy involucra de lectura, procesamiento y escritura.

Archivo de 3 GB:

```
tobias@tobias-XPS-13-9370:/tmp$ curl localhost/minGig > /tmp/gig
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 3072M  100 3072M    0     0   167M      0  0:00:18  0:00:18 --:--:--  110M
```

Caso sin pasar por proxy

```
tobias@tobias-XPS-13-9370:/tmp$ curl localhost/minGig -x socks5://localhost:1080 > /tmp/gig
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 3072M  100 3072M    0     0  98.1M      0  0:00:31  0:00:31 --:--:--  100M
```

Caso pasando por el proxy

Una vez más se puede ver como, el tiempo al pasar por el proxy se convierte en alrededor del doble del caso en que no se usa.

Load Tests

Teniendo en cuenta que uno de los requisitos del trabajo es soportar 500 conexiones concurrentes, fue necesario realizar load tests. Estos permiten, primero verificar que el servidor sea capaz de alcanzar la cantidad de conexiones deseadas (500 en este caso). Luego, permiten analizar el throughput del servidor en base a la cantidad de conexiones que está procesando.

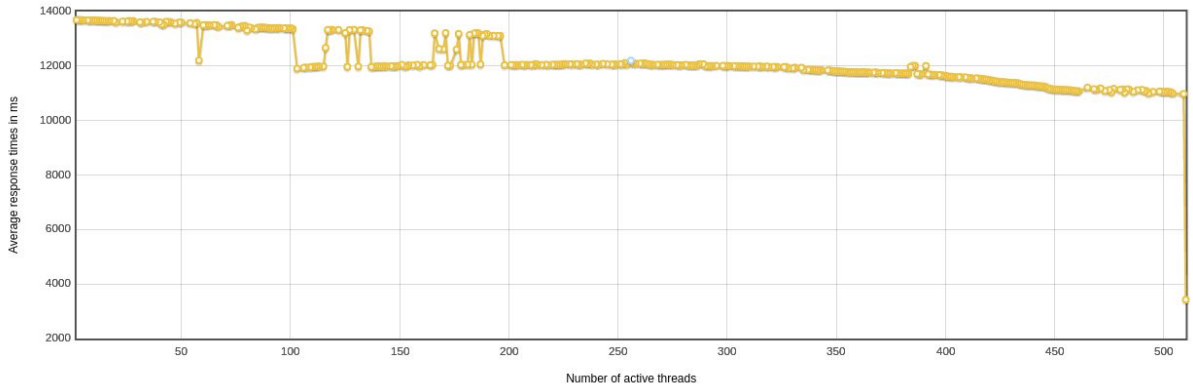

Gráfico Obtenido - Tiempo de Respuesta vs. Cantidad de Conexiones - Sin Ramp Up

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**510**	**1**	**0.20%**	**12202.32**	**3428**	**13697**	**12017.50**	**13575.80**	**13639.00**	**13686.89**	**37.22**	**228246.19**	**3.95**
HTTP Request	510	1	0.20%	12202.32	3428	13697	12017.50	13575.80	13639.00	13686.89	37.22	228246.19	3.95

Resultados Generales del Test - SIn Ramp Up

En esta ocasión, el test realizado consistió en utilizar JMeter junto a un servidor nginx local. Se realizó una prueba de una única vuelta, sin tiempo de ramp up y 510 conexiones HTTP solicitando un archivo de 6MB. El gráfico obtenido permite comparar el tiempo de respuesta de una conexión en función de la cantidad de conexiones ya establecidas en el instante dado.

Observando el gráfico podemos notar que a nuestro servidor le toma alrededor de 12 segundos responder a una consulta, y a medida que aumentan la cantidad de conexiones ya establecidas, el tiempo de respuesta se reduce en un 20%.

Analizando esta información consideramos que el comportamiento del servidor es acorde a lo esperado. Apenas comienza la prueba el servidor recibe consultas hasta alcanzar rápidamente los 510 pedidos de conexión simultáneos, por lo cual, la cantidad de información a procesar inicialmente es muy alta y, como consecuencia, el tiempo de respuesta también lo es. No obstante, a medida que se generan respuestas y la cantidad de información restante por conexión disminuye, el tiempo también se reduce.

Nótese que 1 de los 510 intentos de conexión falló. Esto significa que el límite de sesiones de nuestro proxy es de 509, lo cual es coherente con nuestro análisis previo: 509 sesiones significan 509 sockets cliente y 509 sockets servidor, sumado a los 3 sockets pasivos y 2 file descriptors de salida estándar (stdout y stderr) siempre presentes, obtenemos 1023 sockets abiertos totales en nuestro servidor, prácticamente el limite establecido por pselect.

Esta prueba, más allá de los números concretos, nos sirvió para ver que nuestro server puede realmente responder más de 500 conexiones simultáneas, a pesar de que están se realizan casi simultáneamente, y encontrar el límite exacto del server, que parece ser de 509 sesiones.

Para una prueba de una situación un poco más realista, decidimos repetir el test, pero ahora con un ramp up de 10 segundos y archivos de 12MB (para que las conexiones se mantengan activas por más tiempo). Esto quiere decir que cada 1 segundo, 50 nuevas conexiones se intentarán iniciar con nuestro server.

Los resultados fueron los siguientes:

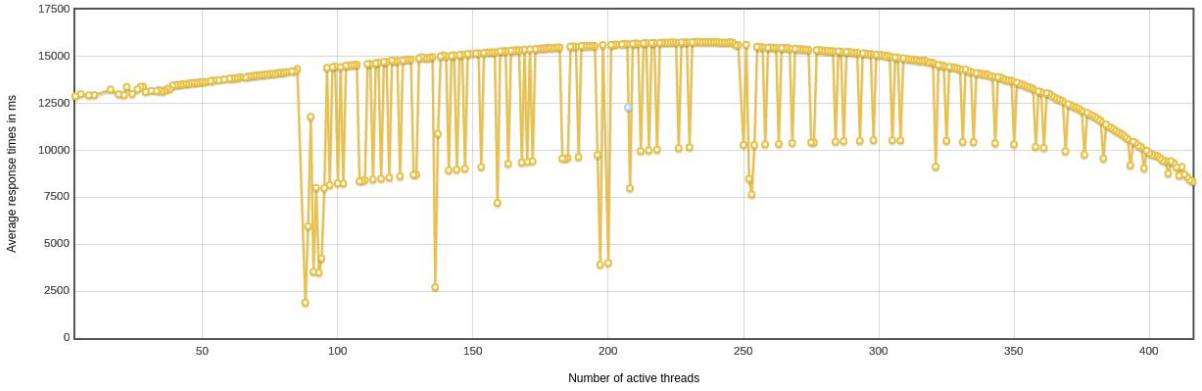

Gráfico Obtenido - Tiempo de Respuesta vs. Cantidad de Conexiones - Con Ramp Up

Requests	Executions			Response Times (ms)							Throughput	Network (KB/sec)	
Label	#Samples	KO	Error %	Average	Min	Max	Median	90th pct	95th pct	99th pct	Transactions/s	Received	Sent
Total	**510**	**0**	**0.00%**	**12298.92**	**1604**	**15798**	**14162.00**	**15621.30**	**15745.25**	**15783.89**	**22.43**	**275618.44**	**2.39**
HTTP Request	510	0	0.00%	12298.92	1604	15798	14162.00	15621.30	15745.25	15783.89	22.43	275618.44	2.39

Resultados Generales del Test - Con Ramp Up

En esta prueba podemos ver que el gráfico cambia un poco de forma, pero sigue siendo coherente con el test anterior. El momento de mayor pedido de conexiones sucede en la mitad del gráfico, cuando ya casi todas las conexiones fueron creadas. A medida que el server logra empezar a responder conexiones, puede resolver más rápido las nuevas, llegando a casi la mitad del tiempo de respuesta con respecto al peor momento.

Podemos ver que inicialmente las respuestas tardan 12.5 segundos, luego en el momento de mayor pedidos de conexión los tiempos de respuesta suben a un poco más de 15 segundos, y ya para el final los tiempos de respuesta están cerca de los 7.5 segundos. A pesar de que los archivos servidos por el nginx son del doble de tamaño, el promedio de respuesta sigue siendo cercano a los 12 segundos. Esto lo atribuimos a que las conexiones simultáneas en general fueron menores, por lo que el server debe tener una mejor performance.

Como era esperable, esta vez ninguna conexión falló, pues nunca se superó el máximo de 509 sesiones simultáneas.

Idealmente, deberíamos tener los resultados de los mismos load tests, pero hechos contra el nginx sin pasar por el proxy. Lamentablemente, no pudimos conseguir ningún resultado confiable. Como el nginx resuelve los pedidos el doble de rápido (o pareciera que más, en condiciones de estrés), la cantidad de conexiones simultáneas no llegaban a ser las suficientes como para armar un gráfico con sentido. Una posibilidad era realizar los mismos tests, pero sirviendo archivos aún más grandes, pero JMeter marcaba que el límite de memoria se había alcanzado. Creemos que para solucionarlo, el test debería ser modificado severamente, por lo que ya tampoco sería razonable compararlo con los 2 tests mostrados previamente.

Ante esta falta de tests de control, entendemos que los resultados de los load tests mostrados aportan un valor mucho menor.

7.Guía de instalación

Para la compilación del proyecto y de los tests se utilizó la herramienta Make de GNU y se utilizó un archivo Makefile.inc que centraliza la elección del compilador (CC) y parámetros de compilación (CFLAGS). Además, en el caso de los archivos fuente se utiliza un segundo archivo .inc para declarar todos los directorios a compilar. Si bien en ciertos casos es necesario agregar entradas manualmente para que sean encontradas en el proceso de compilación, nos permite tener un registro más claro de que se compila y que no.

Por último, respecto al manejo de headers y la inclusión de los mismos, se optó por tomar como path base la carpeta src y referenciar todas las inclusiones respecto a la misma.

En base a lo ya mencionado, para generar el archivo ejecutable del servidor junto al ejecutable del cliente basta con ejecutar el comando make en el directorio principal del proyecto. Para la ejecución basta con ejecutar `./socks5d` o `./pipoClient` respectivamente.

8.Instrucciones para la configuración

Para configurar el server existen 2 formas distintas. Por un lado, el servidor cuenta con una lista de parámetros que pueden ser utilizados al momento de ejecutar el mismo. Los parámetros implementados fueron todos aquellos solicitados por la cátedra con algunos parámetros agregados por nuestra cuenta.

Parámetros agregados

- Deshabilitar clean-up (D)

  Para un mejor manejo de los recursos se implementó un sistema de clean-up para cerrar y liberar recursos ocupados por sesiones inactivas. teniendo en cuenta que esta funcionalidad puede no ser deseable, se ofrece un parámetro de ejecución que permite deshabilitarla.

- Modificar las credenciales de admin. (a)

  Teniendo en cuenta que existe una vía de comunicación secundaria para la configuración del servidor y que los usuarios son los mismos, es necesario que existan dos roles con diferentes niveles de permiso. En base a esto, el servidor inicia la ejecución generando un usuario admin default (admin:papanata). En el caso de querer modificar estas credenciales se ofrece este parámetro donde se setea el nombre y la clave a utilizar para el administrador inicial, separados por ':'.

- Metodo post (doh-method-post)

  Teniendo en cuenta que se implementó tanto GET como POST como métodos válidos para el envío de queries DoH, fue necesario agregar un parámetro que permita modificar el valor default (GET) y utilizar POST.

De todas formas, todos los parámetros agregados fueron incluidos en el man del servidor y en el comando help junto a todos los parámetros solicitados. El archivo man (modificado) provisto puede encontrarse en la raíz del proyecto bajo el nombre “socks5d.8”.

Protocolo SCTP de configuración

Por otro lado, se desarrolló e implementó un protocolo para poder modificar la configuración del servidor en tiempo de ejecución. Este protocolo ofrece Queries para la obtención de información y Modifications para hacer cambios en el servidor.

Si bien el protocolo está especificado en su propio RFC (`pipoProtocol.txt`) y puede utilizarse cualquier herramienta para comunicarse con el servidor, se implementó un cliente que permite aprovechar la totalidad de las funciones ofrecidas por el protocolo.

9.Ejemplos de configuración y monitoreo

A la hora de utilizar el cliente, el proceso se puede dividir en tres partes.

Primero, a la hora de utilizar el ejecutable se puede utilizar con la configuración default, que consiste en utilizar la dirección “127.0.0.1” y el puerto 8080 o, pasar como argumento una dirección IP (v4 o v6) y un puerto. Respetando el orden.

Luego, una vez que se haya establecido la conexión SCTP con el servidor hay que iniciar sesión. Este proceso consiste en ingresar las credenciales de un administrador ya cargado en el servidor.

Por último, se inicia un proceso interactivo para enviar comandos y sus argumentos. Estos son listados la primera vez que se logra establecer de forma exitosa una sesión o utilizando “h” como comando. En caso de que el servidor cierre su lado de la conexión, se percibirá desde el cliente cuando se quiera mandar un nuevo comando. Esto se debe a que el cliente fue implementado de forma bloqueante y, mientras se espera input del usuario, no se tiene información del socket.

10.Documento de diseño del proyecto (que ayuden a entender la arquitectura de la aplicación)

- Dentro de la carpeta src se encuentra todo el código de la aplicación server.
- Dentro de la carpeta test se encuentran los test implementados de src.
- Dentro de src/client se encuentra todo el código relacionado al cliente SCTP de administración.

Módulos del server

- **server.c**: main de la aplicación. Aquí se inicializa el servidor, se llaman a las funciones inicializadores de cada módulo, se crean y cargan los sockets pasivos al selector y se ejecuta el loop principal. También se encarga cada vez que se desbloquea del selector de correr la rutina de limpieza.

- **buffer**: Implementación de un buffer al cual se puede leer y escribir simultáneamente. Clase provista por la cátedra.
- **selector**: Implementación de Selector provista por la cátedra basada en pselect.
- **selectorStateMachine**: Clase para el manejo de una máquina de estados asociada a la clase Selector. Basada en la implementación de la cátedra.
- **reference**: Aquí se encuentran las clases parser y parser_utils provistas por la cátedra. Fueron utilizados en los parsers para identificar strings. Fueron ligeramente modificadas para no depender de memoria dinámica.
- **netutils**: Utilidades referidas a sockets.
- **argsHandler**: Parser de argumentos del servidor provisto por la cátedra con modificaciones menores.
- **statistics**: Clase para el manejo de estadisticas y metricas globales del server.
- **utilities**: Utilidades generales. Se encuentra una implementación propia de base64 y la clase externa khash.h (https://github.com/attractivechaos/klib/blob/master/khash.h, MIT License) para manejo de hash tables.
- **userHandler**: Clase para manejar usuarios de manera volátil. Esta es la única clase que utiliza khash.h y se encuentra abstraída, para no generar vendor lock in.
- **socks5**: dentro de este archivo se encuentran todas las clases relacionadas con el manejo de sesiones en la aplicación.
    - **socks5.c**: lógica relacionada al ciclo de vida de sesiones de usuarios normales del proxy. Lógica de clientes, servers y dns. Funciones manejadoras del read, write y close (frees), y las rutinas de limpieza. Adicionalmente se definen algunos defaults del server, como tamaño de los buffers de I/O.
    - **stateMachineBuilder**: Clase de ayuda para la creación de la selectorStateMachine a partir de las clases State individuales. Sirve para desacoplar socks5.c de la creación de la máquina de estados.
    - **logger**: Simple clase encargada de manejar la lógica de los logs a stdout y stderr no bloqueantes.
    - **administration**: Equivalente a socks5.c pero para las sesiones de administrador. Como es más simple, la máquina de estados asociada está directamente declarada en esta clase. Se encarga del ciclo de vida de las sesiones de administrador.
- **states**: en esta carpeta se encuentran todas las clases encargadas de definir el funcionamiento de cada estado. Hay una carpeta por cada estado, con su mismo nombre. Además, se encuentra la carpeta stateUtilities donde se ubican utilidades usadas por varios estados.

- **parsers**: En esta carpeta se incluyen todos los parsers utilizados en el servidor. Se incluyen helloParser, authRequestParser, dnsParser (tanto query como response), requestParser, spoofingParser (para obtener credenciales) y adminRequestParser (tanto requests como responses del protocolo de administracion que implementamos). Este último tiene varios archivos pues allí también se encuentran las funciones a ejecutar por cada comando del protocolo.

Módulos del client

Todos los archivos mencionados a continuación se encuentran en src/client:

- **client.c**: El mail del cliente. Se encarga de conectarse al servidor y autenticarse. Aquí se encuentra el loop principal del cliente.
- **clientCommandController**: Esta clase se encarga de agrupar el sender, receiver y mensaje de cada comando en un array indexado por número de comando.
- **clientReceivers**: Se definen las funciones que parsean y imprimen los mensajes de respuesta del servidor del protocolo de administración.
- **clientSenders**: Se definen las funciones que construyen y mandan al servidor los mensajes de respuesta del protocolo de administración.

Anexo

Ejemplo de ejecución con DNS mostrando con strace

```
$> curl -x socks5h://localhost:1080 -s tpe.proto.leak.com.ar

Se conecta el cliente
accept(0, {sa_family=AF_INET, sin_port=htons(57030),
sin_addr=inet_addr("127.0.0.1")}, [16]) = 5

Recibe la solicitud
recvfrom(5, "\5\1\0\3\25tpe.proto.leak.com.ar\0P", 512, MSG_NOSIGNAL,
NULL, NULL) = 28

Generación de sockets para comunicarse con el DNS (nginx local)
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) = 6
connect(6, {sa_family=AF_INET, sin_port=htons(8053),
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EINPROGRESS (Operation now in
progress)
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) = 7
connect(7, {sa_family=AF_INET, sin_port=htons(8053),
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EINPROGRESS (Operation now in
progress)

Enviado y recepción de las queries
sendto(6, "GET /dns-query?dns=AAABAAABAAAAA"..., 128, MSG_NOSIGNAL,
NULL, 0) = 128
sendto(7, "GET /dns-query?dns=AAABAAABAAAAA"..., 128, MSG_NOSIGNAL,
NULL, 0) = 128
recvfrom(6, "HTTP/1.1 200 OK\r\nServer: nginx/1"..., 128, MSG_NOSIGNAL,
NULL, NULL) = 128
recvfrom(6, "h: 71\r\nConnection: close\r\ncache-"..., 128,
MSG_NOSIGNAL, NULL, NULL) = 127

Primer intento de conexión a una IP obtenida pero no valida para
establecer conexión.
connect(6, {sa_family=AF_INET, sin_port=htons(80),
sin_addr=inet_addr("240.0.0.1")}, 16) = -1 EINPROGRESS (Operation now in
progress)

Socket generado es liberado por el select y se detecta que salió por un
timeout -> se descarta
getsockopt(6, SOL_SOCKET, SO_ERROR, [ETIMEDOUT], [4]) = 0

Nuevo intento de conexión a una dirección válida
connect(6, {sa_family=AF_INET, sin_port=htons(80),
```

```
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EINPROGRESS (Operation now in
progress)

Levantado por el select sin error -> hay conexión
getsockopt(6, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
```

Strace completo de la conexión:

```
strace: Process 8583 attached
pselect6(5, [0 3 4], [], NULL, {tv_sec=6, tv_nsec=510754000}, {[], 8}) =
0 (Timeout)
pselect6(5, [0 3 4], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1 (in
[0], left {tv_sec=4, tv_nsec=414432400})
accept(0, {sa_family=AF_INET, sin_port=htons(57030),
sin_addr=inet_addr("127.0.0.1")}, [16]) = 5
pselect6(6, [0 3 4 5], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1 (in
[5], left {tv_sec=6, tv_nsec=999994600})
recvfrom(5, "\5\2\0\1", 512, MSG_NOSIGNAL, NULL, NULL) = 4
pselect6(6, [0 3 4], [5], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1 (out
[5], left {tv_sec=6, tv_nsec=999997200})
sendto(5, "\5\0", 2, MSG_NOSIGNAL, NULL, 0) = 2
pselect6(6, [0 3 4 5], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1 (in
[5], left {tv_sec=6, tv_nsec=999996800})
recvfrom(5, "\5\1\0\3\25tpe.proto.leak.com.ar\0P", 512, MSG_NOSIGNAL,
NULL, NULL) = 28
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) = 6
fcntl(6, F_GETFD)                       = 0
fcntl(6, F_SETFL, O_RDONLY|O_NONBLOCK)  = 0
setsockopt(6, SOL_TCP, TCP_SYNCNT, [2], 4) = 0
connect(6, {sa_family=AF_INET, sin_port=htons(8053),
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EINPROGRESS (Operation now in
progress)
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) = 7
fcntl(7, F_GETFD)                       = 0
fcntl(7, F_SETFL, O_RDONLY|O_NONBLOCK)  = 0
setsockopt(7, SOL_TCP, TCP_SYNCNT, [2], 4) = 0
connect(7, {sa_family=AF_INET, sin_port=htons(8053),
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EINPROGRESS (Operation now in
progress)
pselect6(8, [0 3 4], [6 7], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 2
(out [6 7], left {tv_sec=6, tv_nsec=999995700})
getsockopt(6, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
getsockopt(7, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
pselect6(8, [0 3 4], [6 7], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 2
(out [6 7], left {tv_sec=6, tv_nsec=999996500})
```

```
sendto(6, "GET /dns-query?dns=AAABAAABAAAAA"..., 128, MSG_NOSIGNAL,
NULL, 0) = 128
sendto(7, "GET /dns-query?dns=AAABAAABAAAAA"..., 128, MSG_NOSIGNAL,
NULL, 0) = 128
pselect6(8, [0 3 4 6 7], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1
(in [6], left {tv_sec=6, tv_nsec=920014400})
recvfrom(6, "HTTP/1.1 200 OK\r\nServer: nginx/1"..., 128, MSG_NOSIGNAL,
NULL, NULL) = 128
pselect6(8, [0 3 4 6 7], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 2
(in [6 7], left {tv_sec=6, tv_nsec=999995700})
recvfrom(6, "h: 71\r\nConnection: close\r\ncache-"..., 128,
MSG_NOSIGNAL, NULL, NULL) = 127
close(6)                                = 0
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) = 6
fcntl(6, F_GETFD)                       = 0
fcntl(6, F_SETFL, O_RDONLY|O_NONBLOCK)  = 0
setsockopt(6, SOL_TCP, TCP_SYNCNT, [2], 4) = 0
connect(6, {sa_family=AF_INET, sin_port=htons(80),
sin_addr=inet_addr("240.0.0.1")}, 16) = -1 EINPROGRESS (Operation now in
progress)
recvfrom(7, "HTTP/1.1 200 OK\r\nServer: nginx/1"..., 128, MSG_NOSIGNAL,
NULL, NULL) = 128
pselect6(8, [0 3 4 7], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1 (in
[7], left {tv_sec=6, tv_nsec=999995800})
recvfrom(7, "h: 97\r\nConnection: close\r\ncache-"..., 128,
MSG_NOSIGNAL, NULL, NULL) = 128
pselect6(8, [0 3 4 7], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1 (in
[7], left {tv_sec=6, tv_nsec=999996900})
recvfrom(7, "one\3008^\273B\f\0\0\16\20\0\0\2X\0\t:\200\0\0\1,", 128,
MSG_NOSIGNAL, NULL, NULL) = 25
pselect6(8, [0 3 4 7], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1 (in
[7], left {tv_sec=6, tv_nsec=999997100})
recvfrom(7, "", 128, MSG_NOSIGNAL, NULL, NULL) = 0
close(7)                                = 0
pselect6(7, [0 3 4], [6], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1 (out
[6], left {tv_sec=3, tv_nsec=868251400})
getsockopt(6, SOL_SOCKET, SO_ERROR, [ETIMEDOUT], [4]) = 0
close(6)                                = 0
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP) = 6
fcntl(6, F_GETFD)                       = 0
fcntl(6, F_SETFL, O_RDONLY|O_NONBLOCK)  = 0
setsockopt(6, SOL_TCP, TCP_SYNCNT, [2], 4) = 0
connect(6, {sa_family=AF_INET, sin_port=htons(80),
sin_addr=inet_addr("127.0.0.1")}, 16) = -1 EINPROGRESS (Operation now in
progress)
pselect6(7, [0 3 4], [6], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1 (out
```

```
[6], left {tv_sec=6, tv_nsec=999993800})
getsockopt(6, SOL_SOCKET, SO_ERROR, [0], [4]) = 0
pselect6(7, [0 3 4], [5], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1 (out
[5], left {tv_sec=6, tv_nsec=999994200})
sendto(5, "\5\0\0\1\0\0\0\0\0\0", 10, MSG_NOSIGNAL, NULL, 0) = 10
stat("/etc/localtime", {st_mode=S_IFREG|0644, st_size=1100, ...}) = 0
write(1, "2020-11-13T00:59:22Z\tanonymous\tA"..., 76) = 76
write(1, "", 0)                         = 0
pselect6(7, [0 3 4 5 6], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1
(in [5], left {tv_sec=6, tv_nsec=999994800})
recvfrom(5, "GET / HTTP/1.1\r\nHost: tpe.proto."..., 512, MSG_NOSIGNAL,
NULL, NULL) = 85
pselect6(7, [0 3 4 5 6], [6], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1
(out [6], left {tv_sec=6, tv_nsec=999995400})
sendto(6, "GET / HTTP/1.1\r\nHost: tpe.proto."..., 85, MSG_NOSIGNAL,
NULL, 0) = 85
pselect6(7, [0 3 4 5 6], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1
(in [6], left {tv_sec=6, tv_nsec=999996000})
recvfrom(6, "HTTP/1.1 200 OK\r\nServer: nginx/1"..., 512, MSG_NOSIGNAL,
NULL, NULL) = 266
pselect6(7, [0 3 4 5 6], [5], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1
(out [5], left {tv_sec=6, tv_nsec=999995600})
sendto(5, "HTTP/1.1 200 OK\r\nServer: nginx/1"..., 266, MSG_NOSIGNAL,
NULL, 0) = 266
pselect6(7, [0 3 4 5 6], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1
(in [5], left {tv_sec=6, tv_nsec=999996600})
recvfrom(5, "", 512, MSG_NOSIGNAL, NULL, NULL) = 0
shutdown(6, SHUT_WR)                    = 0
pselect6(7, [0 3 4 6], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 1 (in
[6], left {tv_sec=6, tv_nsec=999994900})
recvfrom(6, "", 512, MSG_NOSIGNAL, NULL, NULL) = 0
shutdown(5, SHUT_WR)                    = 0
close(6)                                = 0
close(5)                                = 0
pselect6(5, [0 3 4], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 0
(Timeout)
pselect6(5, [0 3 4], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}) = 0
(Timeout)
pselect6(5, [0 3 4], [], NULL, {tv_sec=7, tv_nsec=0}, {[], 8}
```