Instituto Tecnológico de Buenos Aires

Protocolos de Comunicación

Trabajo Práctico Especial

1º cuatrimestre 2021

Grupo 7

AUTOR	LEGAJO
Serpe, Octavio Javier	60076
Rodríguez, Manuel Joaquín	60258
Arca, Gonzalo	60303
Parma, Manuel Félix	60425

Fecha de entrega: 15/06/2021

Índice

Descripción detallada de los protocolos y aplicaciones desarrolladas

---

Proxy HTTP

Lista de conexiones actuales

Implementamos una lista doblemente encadenada de una estructura `connection_node` desarrollada por nosotros que incluye información perteneciente a una conexión abierta actualmente. Se decidió por la doble encadenación para el rápido agregado y borrado de nodos, además de que una lista es útil para iterar por las distintas conexiones cuando se activa un socket para leer o escribir.

Nodo de conexión

La lista descrita en el punto anterior utiliza una estructura `connection_node` que incluye (además de los punteros `*next` y `*previous`) otra estructura `connection_data` con la información pertinente a la conexión. En ella se encuentran los sockets de cliente y servidor, los buffers para ambas transferencias, estados de conexión y de parseo, lista de direcciones para intentar la conexión al servidor, información relacionada a las consultas DoH, y un timestamp para evitar conexiones inactivas.

Gracias a este encapsulamiento de los datos, es más simple el pasaje de parámetros entre funciones que necesitan o modifican información de una conexión, además de que mantenemos consistente y en un único lugar esta información.

Manejo de errores

Cuando surgen errores dentro del funcionamiento del proxy, como puede ser por ejemplo que no se resuelva el dominio, el proxy se encargará de poner en el buffer destinado a la escritura hacia el cliente el mensaje de error apropiado. Y cerrará la conexión una vez que se asegura que fue correctamente enviado.

Manejo de escritura/lectura de cliente y servidor

La lectura tanto del cliente como del servidor siempre se mantiene abierta salvo en el caso que los buffers destinados a almacenar esta información se encuentren llenos. Esto sucede en el caso en el que la otra punta no esté leyendo nada del mismo, una vez lea algo se volverá a atender la lectura de la punta que había sido bloqueada.

Por el otro lado, la escritura hacia uno de los objetivos solo se activa su par envío información. Y se desactiva cuando esta información fue enviada en su completitud.

Este flujo nos permite no atender de forma innecesaria a uno de *peers* y así mantener una ejecución eficiente en el programa.

Uso de 2 sockets pasivos para IPv4 e IPv6

Con el fin de poder escuchar tanto a direcciones de IPv4 e IPv6 en el proxy y en el servidor de monitoreo, se crearon 2 sockets pasivos para cada uno de éstos, uno para IPv4, y el otro para IPv6. Por defecto se escucha en ambos sockets. Una limitación interesante de ésta configuración es la imposibilidad de escuchar por IPv4 e IPv6 en simultáneo cuando las direcciones a escuchar son especificadas por argumento, ya que se parsean y se detecta si son direcciones IPv4 o IPv6, y a continuación se crea un único socket dependiendo del caso.

Registros y STDOUT no bloqueante

El proxy cuenta con un manejo de registro tanto de ACCESO como de PASSWORD, donde el primero registra *fecha*, *tipo*, *ip origen*, *puerto origen*, *metodo*, *target (con puerto destino)*, *status code*. Un ejemplo de este registro podría ser

```
2021-06-15T14:26:13Z   A    [127.0.0.1]:9955   GET    foo.leak.com.ar:80    200
```

donde por comodidad las ips se encierran entre '`[`' y '`]`' para separarlas del puerto.

El registro de `PASSWORD` registra *fecha*, *tipo*, *protocolo*, *target*, *puerto destino*, *usuario*, *contraseña*. Es importante aclarar que decidimos únicamente sniffear las credenciales HTTP de tipo BASIC. Con el proxy corriendo y curl utilizándolo se puede ejecutar

```
curl http://user:password@foo.leak.com.ar
```

y como respuesta se obtendría

```
2021-06-15T16:03:55Z        P    HTTP   foo.leak.com.ar:80  user:password
```

Por otro lado, en el caso de solicitar un `CONNECT` para utilizar POP3 los registros de `PASSWORD` difieren en cuanto a la representación de las credenciales, donde se encierra tanto al usuario como a la contraseña entre '`<`' y '`>`' para poder identificarlas correctamente, dado que no hay carácter delimitador como en HTTP con credenciales Basic. Con el proxy corriendo, *netcat* utilizándolo y corriendo un *server* POP3 se puede ejecutar (en nuestro caso utilizamos *Dovecot* como *server* POP3 en el puerto 110 y credenciales `<manurodriguez>:<12345678>`)

```
Connection to localhost 8080 port [tcp/http-alt] succeeded!
CONNECT 127.0.0.1:110 HTTP/1.1
HTTP/1.1 200 Connection Established

+OK Dovecot (Ubuntu) ready.
USER manurodriguez
+OK
pass 12345678
+OK Logged in.
```

y como respuesta se obtendría

```
2021-06-15T17:33:09Z    P       [127.0.0.1]:110    <manurodriguez>:<12345678>
```

Dichos registros son visualizados mediante el file descriptor de `STDOUT` en medio de la ejecución del proxy mismo, es por ello que optamos por hacer `STDOUT` no bloqueante. De esta manera garantizamos que una interrupción por escritura o falla a `STDOUT` no inutiliza al proxy, y en consecuencia, la visualización de los registros corre el riesgo de retrasarse y no ser instantánea.

Parser HTTP

Poca restricción sobre métodos, '//', espacios de más entre campos y versión

Decidimos solo restringir los métodos por su longitud, la cual deberá ser menor o igual a 24 caracteres, para dar libertad a los usuarios de utilizar cualquiera de todos los métodos existentes en el RFC o los que implementen ellos. Es importante aclarar que los métodos `CONNECT` y `OPTIONS` respetan el comportamiento descrito en el RFC 7231[1], esto quiere decir que si el proxy recibe el método connect (con variaciones de minúsculas y mayúsculas, y análogamente con el método options) el proxy no garantiza el correcto funcionamiento y por lo tanto podría recibir una respuesta NO deseada el cliente.

También decidimos ser pocos restrictivos sobre las dos barras que deberían aparecer dentro de la URI HTTP. Simplemente se saltean todas las '`/`' luego del '`schema:`'. De esta manera el cliente puede decidir si incluir como no las mismas.

El parser se encarga de descartar los múltiples espacios que pueden existir entre los distintos campos de la *start line* (comportamiento que se debería seguir según el RFC 7230[2]).

En todas las requests salientes la versión a utilizar será HTTP/1.0, al parser solo le interesa que exista al menos un carácter (distinto del espacio) en este campo. Se

[1] https://datatracker.ietf.org/doc/html/rfc7231
[2] https://datatracker.ietf.org/doc/html/rfc7230#section-3.1.2

decidió por la versión 1.0 pues no maneja comunicaciones persistentes, aunque es cierto que el *origin server* objetivo puede responder en la versión que él quiera por lo que este comportamiento no se puede asumir.

Se envían la startline y los headers una vez se completa cada uno

Cuando la máquina no se encuentra parseando el *body*, las líneas se copian a la salida solo cuando se encuentran completas, es decir, cuando se recibe el `CRLF` pertinente. Esta decisión se tomó ya que para el servidor es inútil tener la *start line*, o algún *header* incompleto, además de que son campos donde les impusimos una longitud limitada y no muy extensa. De esta manera el proxy no pierde tiempo enviando de a *chunks*.

Estructura HTTP parser

Cada nodo de conexión contiene dentro una estructura destinada al parser HTTP. Esta estructura dentro tiene un campo destinado a una estructura contenedora de los posibles campos POP3 (que se crearán según sea necesario), una estructura con todos los datos que utiliza la máquina de estados HTTP y una estructura donde se guardará toda la información relevante sobre el *request*. Guardar esta información nos permite, por ejemplo, acceder desde fuera del parser a donde se encuentra el destino y si es un dominio, una dirección IPv4 o una dirección IPv6. Así como también nos permite saber si la *request* fue inválida, o si requiere el tratamiento de un `CONNECT` o POP3, entre otras cosas.

La estructura con datos de la máquina de estados contiene el estado actual de la misma, un índice que permite retomar la copia sobre una posición específica de un buffer buscado, el estado de la búsqueda del objetivo (para saber si ya se puede realizar la consulta DNS o no), un buffer que contiene la respuesta y un estado que indica sobre qué estado de parseo del formato HTTP se encuentra la maquina (si esta leyendo la *start line*, los *headers*, o el *body*).

Manejo de CONNECT

Una vez que en el request HTTP se identifica el origin server objetivo, si este pertenece al método connect (con un puerto destino distinto al 110) se lo dejara de parsear y simplemente se pasará a un estado de forwardear toda la información punta a punta.

En el caso que sea con puerto 110, significa que utiliza el protocolo POP3. Como en este caso queremos encontrar las credenciales de acceso se seguirá parseando la comunicación hasta que se logre esto y se garantice la validez de las mismas. Luego, pasará a tener el comportamiento ya explicado del `CONNECT`.

Identificación de credenciales en HTTP y conexión instantánea

El parser reconoce dos headers de su interés: Authorization y Host. Respecto al primero, en caso de encontrarlo y que el tipo de las credenciales sea Basic, mediante una librería para *base64* que utilizamos como decoder, se cargan los respectivos valores.

Respecto al segundo header, en el caso de no haber resuelto el target, instantáneamente apenas se lo encuentra, la máquina de estados sale y deriva la conexión en el proxy para realizar la misma lo más temprano posible (así en caso de error no se habrá parseado toda la request innecesariamente). Análogamente sucede lo mismo en el caso de recibir una URI con path absoluto.

Parser POP3

Estructura POP3 parser

La estructura que almacena el nodo mantiene un parser de los comandos, un parser de las respuestas y un contador que indica cuántas líneas se enviaron y todavía no se respondieron. Este contador nos permite saber cuándo se debería encontrar la contraseña, pues en dicho momento el valor de esta variable es cero gracias a que el parseador de los comandos finaliza su ejecución, momentáneamente, apenas completa un par usuario-contraseña.

Separación parser de command y de response

Se utilizaron dos parsers distintos para el manejo del protocolo POP3, en el caso que se realice un `CONNECT` a un puerto 110, pues las *requests* (*commands*) y las respuestas seguían un formato distinto. Por otro lado, consideramos que ambas máquinas de estado eran pequeñas y si se realizaba todo en una sola iban a confundirse conceptos, aumentar una complejidad evitable y código confuso.

Una vez se encuentran credenciales (un par *user-password*), se finaliza la ejecución del parser de la *request* esperando la verificación por lado del servidor. En caso de error, se buscarán unas nuevas credenciales. En caso que las credenciales sean inválidas, se retoma la ejecución tanto de la máquina que parsea la respuesta como el comando POP3. Dicha ejecución continúa hasta que el usuario ingrese correctamente las credenciales, y se pasará al estado del método CONNECT normal (sin POP3), evitando el sniffeo de los mismos.

DoH Requests

Estructura DoH Data

Se creó una estructura `doh_data` para almacenar información sobre las consultas DoH de una conexión a abrir. Esta estructura aparece dentro de la estructura `connection_data`. Los campos en `doh_data` incluyen al file descriptor del socket asignado a la conexión con el server DoH, una variable de estado de la consulta, un buffer para almacenar la consulta y respuesta, la longitud de la respuesta, un array con los tipos de consulta DNS a realizar (en nuestro caso A y AAAA), y un contador para manejar las consultas de estos tipos.

DoH States y Codes

Se utilizaron definiciones de `enum` para denominar códigos de estado del envío y parseo de las consultas y respuestas DoH (`doh_send_status_code` y `doh_parser_status_code` respectivamente), y los diferentes estados del flujo de resolución DoH (`doh_state`). Los códigos se utilizan para manejar casos de error y éxito en el retorno de distintas funciones, mientras que el estado sirve para indicar las acciones a tomar según lo enviado/leído.

Estructuras de Header, Question y Request DoH

Para guardar los datos literales de las consultas DoH y DNS, armamos las estructuras `dns_header`, `dns_question` y `http_dns_request`; `dns_header` utiliza definición de bytes para no ocupar espacio extra en memoria y representa el header de una consulta DNS, `dns_question` incluye un puntero a un *hostname* a resolver y el *type* y *class* de la consulta, y `http_dns_request` se utiliza para guardar la información del *header* HTTP utilizado. Se decidió implementar únicamente una consulta de método HTTP `POST`, ya que al no tener que codificar la consulta DNS en el cuerpo del paquete nos permite un rápido y fácil armado de consultas y parseo de las respuestas.

dohclient.c

En `dohclient.c` se encuentran las funciones que se llaman desde otras partes del proyecto para manejar las consultas DoH. La función `connect_to_doh_server()` se encarga de abrir la conexión al servidor DoH en primer lugar. Nuestra implementación utiliza una conexión para cada cliente del proxy, ya que esta se va a cerrar en cuanto obtengamos la respuesta DoH por lo que consideramos que no afecta a la performance del proxy. La siguiente función en el flujo es `handle_doh_request()`, que se encarga de armar y enviar la consulta DoH en base al hostname a resolver. Dentro de esta función se llama a `dohsender`. Por último, `handle_doh_response()` se encarga de recibir la respuesta DoH y llamar a las funciones de parseo de

`dohparser.c`. Ya que el mensaje puede no llegar completo, los estados doh_state manejan las distintas partes del parseo. También se encuentran `is_connected_to_doh()` y `check_requests_sent()`, funciones booleanas para manejar casos chequeados.

dohsender.c

Este archivo encapsula el comportamiento de la preparación y el envío de las consultas DoH al servidor que las resuelve. La función `prepare_doh_request()` utiliza los datos seteados en las estructuras de `doh_data` junto con funciones auxiliares estáticas para copiar a un buffer la consulta, mientras que `send_doh_request()` envía los datos de este buffer utilizando la función `send()`.

dohparser.c

Este archivo contiene las distintas funciones que se encargan de parsear las secciones de una consulta DoH. Además de guardar la información obtenida, se encarga de comprobar que los datos recibidos sean consistentes con la consulta enviada. Las posibles IPs que resuelven al host indicado se guardan en una lista del tipo `addr_info_node` dentro de un `connection_node` que utiliza una `union` para guardar `sockaddr_in` y `sockaddr_in6` alternativamente.

dohutils

`dohutils.c` incluye funciones misceláneas que se utilizan en los archivos previamente indicados relacionados con DoH. La función `setup_doh_resources()` se encarga de alocar el espacio para la información utilizada en un `connection_node` para resolver DoH; `add_ip_address()` se encarga de agregar una nueva IP a la lista de *addresses* dentro de un nodo; y `free_doh_resources()` se llama para liberar los recursos alocados por `setup_doh_resources()` luego de que se hayan finalizado las consultas DoH.

PCAMP - Proxy Configuration and Monitoring Protocol

El protocolo

En el archivo `especificaciones-pcamp.pdf`, que se encuentra en la carpeta raíz del proyecto, se describe en detalle el protocolo de monitoreo y configuración diseñado por nosotros y utilizado en este proyecto. A continuación se explica la implementación del servidor y cliente para utilizar este protocolo.

PCAMP Client

El archivo `pcampclient.c` contiene el código del programa utilizado para configurar y monitorear el servidor proxy utilizando el protocolo PCAMP como cliente. Dicho programa posee una interfaz de línea de comando básica y pide al usuario si desea realizar una consulta de una *query* o un cambio de configuración del proxy. Internamente, utilizamos funciones auxiliares estáticas para armar el *request* PCAMP, enviarlo, y luego recibir una respuesta y parsearla. Ya que el protocolo PCAMP utiliza UDP para conectarse al proxy, y el mismo servidor utiliza una política de mejor esfuerzo a la hora del manejo de la recepción y envío de las *requests* (detallado en sección 5.1. del documento de especificaciones del protocolo), se hizo uso de un timeout y de retransmisión con el fin de manejar casos de envío o recepción erróneos y de ausencia de respuesta de parte del servidor. Adicionalmente, se utiliza una estructura auxiliar `pcamp_response_info` para guardar información de la respuesta obtenida; en esta estructura se implementa una `union` para poder guardar alternadamente información de una *query* o una configuración.

PCAMP Server

El archivo `pcampserver.c` implementa las funciones que el proxy utiliza para escuchar y atender las consultas PCAMP de *query* y configuración: `setup_pcamp_sockets()` crea y guarda los file descriptors de dos sockets UDP para IPv4 y IPv6 que van a escuchar las requests PCAMP, y `handle_pcamp_request()` se encarga de atender a uno de estos sockets que haya sido activado durante la espera del `select()`. Está última función también se encarga de guardar la *response*, parsearla, realizar el cambio o *query* pedido, y devolver un response PCAMP al cliente de origen; todas estas operaciones se realizan llamando a funciones estáticas dentro del archivo. Utilizando el campo `AUTHORIZATION` del protocolo PCAMP se implementó un sistema simple de *digest access authentication*, basado en el intercambio de una *passphrase* hasheada con el objetivo de que el servidor de PCAMP no realice ninguna operación cuando el hash SHA-256 de la contraseña interna del servidor no coincida con la del *request* PCAMP.

PCAMP Utils: estructuras y estados

Este archivo incluye una única función: `sha256_digest()`, que se encarga de hashear un string utilizando el algoritmo SHA-256. Por otro lado, `pcamputils.h` define varias estructuras y constantes utilizadas en los dos archivos descritos anteriormente. Por un lado, se definieron varios `enum` para diferenciar los tipos de métodos PCAMP (`pcamp_method`), el tipo de paquete (`pcamp_op`), los tipos de *query* (`query_type`), los tipos de configuraciones (`config_type`) y los códigos de estado

(pcamp_status_code). Luego, se utilizan estructuras auxiliares para guardar valores de los *headers*, *requests* y *responses* PCAMP, y una última estructura pcamp_request_info que utiliza una union para guardar alternados una *query* o configuración PCAMP en base a la *request* utilizada.

Otros

connection.h

En este *header* se describen las estructuras previamente explicadas para los nodos de conexiones (connection_node y connection_data), además de una estructura para el *header* de la lista de conexiones (connection_header). Este *header* incluye una referencia al primer y último elemento de la lista, el file descriptor máximo actualmente en uso, la cantidad de clientes actuales, una subestructura de estadísticas para las *queries* de monitoreo y un buffer para volcar en salida estándar.

Por otro lado, se definen funciones para manejar conexiones: setup_connection_resources() crea un connection_node que, una vez realizada la conexión, se debe agregar a la lista de conexiones mediante add_to_connections(); close_server_connection() cierra la conexión del proxy al servidor destino, mientras que close_connection() se encarga de cerrar la conexión con el cliente; close_buffer() es una función auxiliar para liberar los recursos de un buffer utilizado para el intercambio de información en la conexión; y las funciones indicadas con "pop3" en su nombre se encargan de alocar y liberar recursos dentro de un connection_node para el parseo de mensajes POP3.

netutils.h

Se definieron funciones auxiliares para utilidades de la red en este archivo: parse_ip_address() y parse_port() se encargan de verificar y convertir a una variable utilizable los strings de IP y puerto respectivamente; strcmp_case_insensitive() implementa una comparación de strings pero case insensitive; hton64() y ntoh64() fueron implementadas para el manejo de variables de 64 bits en big-endian ya que las librerías aceptadas en la versión de POSIX contra la cual se trabajó solo implementan para 16 y 32 bits. Finalmente, copy_from_buffer_to_buffer() para copiar información entre buffers del tipo definido en buffer.h

logger.h

Librería que provee dos macros: logger() y logger_peer(). La primera posee 4 niveles: INFO, DEBUG, ERROR y FATAL, y escribe en STDERR. Dicho file descriptor no se hizo bloqueante pues logger finaliza la ejecución ante el nivel FATAL y antes imprime el respectivo mensaje, informando la línea, archivo y función donde se

la llamó (y sólo se terminó utilizando este nivel en el código). Por otro lado, `logger_peer()` se encarga de realizar lo mismo que `logger()` pero no posee niveles, recibe de qué peer se trata (SERVER o CLIENT) y antepone el mismo al mensaje. Nuevamente esta escritura se realiza a STDERR.

args.h y buffer.h

Utilizamos los archivos `args.h` y `args.c` provistos por la cátedra para implementar `proxyargs.c` y `pcampargs.c`, que se encargan de parsear y guardar los argumentos de ejecución del proxy y del cliente de monitoreo respectivamente. Estos datos se almacenan en estructuras globales para que puedan ser accedidas desde los archivos necesarios en el proyecto. También se utilizó la librería de buffer provista por la cátedra para el guardado de información.

Problemas encontrados durante el diseño y la implementación

Proxy HTTP

Garbage collector

Nos encontramos con la situación en la que si nos intentamos conectar a un destino que no nos aceptaba, ni nos mandaba rechazo, caiamos en la espera de la finalización del timeout default del método *connect*. Para evitar esta larga espera, implementamos un timeout de 10 segundos en el cual si el cliente no se conecta, se probará con la siguiente dirección posible en caso de existir. Caso contrario, se descarta a este cliente.

También, esta implementación de timeout nos permitió cerrar conexiones que permanecieron durante este mismo tiempo sin realizar ningún tipo de interacción punta a punta, evitando así posibles conexiones persistentes.

Parser HTTP

Buffer auxiliar para parsers

Al utilizar las máquinas de estados que consumen carácter por carácter, nos encontramos que requerimos de otro buffer donde volcar esta misma información (que podría tener un formato distinto o no según el parser) para poder luego enviarla al servidor.

A la hora de finalizar el body no hay delimitador que lo indique, por lo tanto luego de terminar de parsear los headers se procede a igualar punteros para continuar la ejecución directamente del buffer parseado donde previamente se cargó el contenido posterior a los headers.

Parser POP3

El protocolo POP3 posee un mecanismo de request - response, pero el primero en dar el OK (mensaje de bienvenida) es el origin server, y dado que para saber si las credenciales ingresadas son correctas pedimos que la diferencia en número de request y response sea cero, por lo tanto en realidad es el número de responses quien se adelanta al número de requests, lo cual se solucionó inicializando el número de responses en uno, y ante la respuesta se resta la cantidad de líneas (CRLF) leídas, de esta manera ambos arrancan en cero luego del mensaje de bienvenida.

DoH Requests

Doble request

Para la resolución de hostnames, inicialmente se había implementado el envío de una única request DoH con dos “Questions” dentro: una tipo A para IPv4, y otra tipo AAAA para IPv6. Pero nos encontrábamos con que las respuestas del servidor DoH sólo incluían una respuesta (generalmente la de tipo A), cuando utilizando el comando dig se resolvían IPs de ambos tipos para ciertos dominios (como google.com). Al investigar nos encontramos que no es estándar el uso de varias queries/questions dentro de un request DNS. Por esta razón, decidimos implementar el envío de dos paquetes request DoH para cada uno de los dos tipos. Primero enviamos el tipo A, esperamos la respuesta, para luego enviar de tipo AAAA y obtener la respuesta de este. Todas las IPs obtenidas de las responses se agregan a la lista de posibles addresses del Connection Node actual.

Limitaciones de la aplicación

Solo se acepta encoding en Basic para el Authorization

Por más que nos hubiese gustado soportar otras posibles codificaciones, entendemos que no es el objetivo de esta materia ni proyecto por lo que se decidió que con Basic sería suficiente.

Argumento --doh-query está inutilizado

Ya que únicamente se implementó consulta DoH con el método POST, este parámetro de ejecución no se utiliza para realizar esta consulta. De todas maneras, se acepta como parámetro (como indica el manual) y se almacena para posibles implementaciones futuras.

Warning de Valgrind cuya solución desconocemos

Al correr el análisis dinámico en ejecución con Valgrind obtenemos el siguiente warning:

```
Syscall param socketcall.bind(my_addr.sin6_scope_id) points to
uninitialised byte(s)
```

Leyendo el manual de Linux para *ipv6*(7) descubrimos que se trata de un elemento de la estructura `sockaddr_in6` que estamos iniciando en 0. El *warning* parece provenir del hecho de que no es una dirección de memoria válida. El manual de Linux indica que `sockaddr_in6` debe estar seteado en un "*interface index*" referenciado en el manual de *netdevice*(7). Investigando, encontramos que una posible solución es utilizar la función del manual getifaddrs(3) para obtener una lista de interfaces y luego iterar por cada una comparando con la IP a utilizar para obtener el valor necesitado, pero no llegamos a implementarlo por cuestión de tiempo.

Llamado recursivo al proxy

Se buscó realizar un manejo consciente de cuando un cliente solicita un recurso con la dirección objetivo siendo la misma sobre la cual corre el proxy pero nos resultó muy complejo. Esto se debe a que habían demasiados casos a considerar, ya que no solo era chequear por localhost si no también por las ips que resolvían los dominios, que podría llevar a localhost. También, el proxy podría correr en una ip distinta de localhost por lo que ese sería otro caso a considerar.

Posibles extensiones

A continuación se enumeran distintas adiciones y modificaciones al proyecto que consideramos interesante para una hipotética segunda versión del proxy:

1. Aceptar más tipos de codificaciones distintas a Basic en el header Authorization.
2. Utilizar Content-Length de HTTP en el parseo
3. Implementar consultas DNS over HTTP**S**.

4. Utilizar una única conexión persistente al servidor DoH para realizar consultas DNS.
5. Implementar consultas DoH que utilicen el método HTTP `GET`.
6. Solucionar warning sobre `sin6_scope_id` de Valgrind.

Conclusiones

Principalmente pudimos comprobar la dificultad de mantener y hacer estable un proxy, en este caso HTTP, y que el mismo cumpla con los estándares indicados en los RFCs. También, notamos la complejidad que conlleva la creación de un protocolo y que se debe tener un buen manejo de comunicaciones en red y de manejo de bytes.

Encontramos que el proxy desarrollado mantiene un *throughput* muy alto, que lo pudimos comprobar corriendo varios tests de velocidad (*www.fast.com*) en simultáneo y verificando que éstos otorgaban un ancho de banda resultante similar al obtenido sin el proxy.

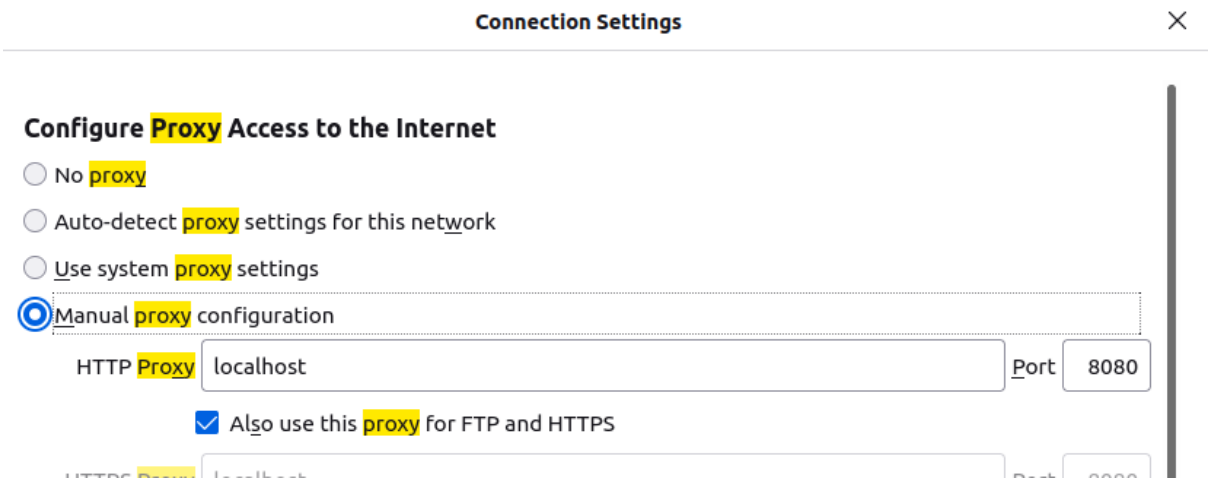
Por otro lado, nos asombró la simpleza y facilidad con la cual se pueden obtener credenciales de los usuarios que desconocen lo que sucede por detrás de una página no segura y envían su información.

Ejemplos de prueba

Ejecutando el programa sin especificar un puerto, se correrá en el puerto 8080 por default. Siguiendo esto, un ejemplo fácil con `curl` seria:

```
manurodriguez@ACERNOTEBOOK:~$ ALL_PROXY=localhost:8080 curl http://foo.leak.com.ar/
hola mundo!
manurodriguez@ACERNOTEBOOK:~$ ALL_PROXY=localhost:8080 curl -v http://foo.leak.com.ar/
* Uses proxy env variable ALL_PROXY == 'localhost:8080'
*   Trying 127.0.0.1:8080...
* TCP_NODELAY set
* Connected to localhost (127.0.0.1) port 8080 (#0)
> GET http://foo.leak.com.ar/ HTTP/1.1
> Host: foo.leak.com.ar
> User-Agent: curl/7.68.0
> Accept: */*
> Proxy-Connection: Keep-Alive
>
* Mark bundle as not supporting multiuse
< HTTP/1.1 200 OK
< x-amz-id-2: SKHMkwNQUYVlWLzl2M9BP3P1Vz5YZpa6KqA2BxUVbEtdgxjviI0Q3YAii7Vjec+w8txz/pHFZeM=
< x-amz-request-id: 7FD9DKH3XR318S2S
< Date: Tue, 15 Jun 2021 20:41:25 GMT
< Last-Modified: Tue, 23 Aug 2016 00:13:08 GMT
< ETag: "bd363b1d3c4272fe93618c8fdac2435a"
< Content-Type: text/html
< Content-Length: 12
< Server: AmazonS3
< Connection: close
<
hola mundo!
* Closing connection 0
```

Para utilizarlo en un *user agent*, se debe configurar el proxy que utiliza el mismo y setear allí el proxy HTTP del proyecto. En el caso específico de Firefox seria:



Guía de instalación

Dependencias del proyecto

- make
- Compilador de C (conforme con estándares C11 y POSIX.1-2001)
- Librerías criptográficas de OpenSSL

Cómo compilar el proyecto

Situarse dentro de la carpeta del proyecto y ejecutar el siguiente comando

```
$ make all
```

Cómo ejecutar el proyecto

Para correr el proxy HTTP:

```
$ ./httpd
```

Para correr el cliente del protocolo de monitoreo

```
$ ./httpdctl
```

Instrucciones para la configuración

Argumentos de línea de comando para httpd

`--doh-ip dirección-doh`
Establece la dirección del servidor DoH. Por defecto `127.0.0.1.`

`--doh-port port`
Establece el puerto del servidor DoH. Por defecto 8053.

`--doh-host hostname`
Establece el valor del header Host. Por defecto `localhost`.

`--doh-path path`
Establece el path del request DoH. por defecto `/getnsrecord`.

`--doh-query query`
Establece el query string si el request DoH utiliza el método Doh por defecto ?dns=.

`-h`

Imprime la ayuda y termina.

`-N`
Deshabilita los *password dissectors*.

`-l dirección-http`
Establece la dirección donde servirá el proxy HTTP. Por defecto escucha en todas las interfaces.

`-L dirección-de-management`
Establece la dirección donde servirá el servicio de management. Por defecto escucha únicamente en loopback.

`-o puerto-de-management`
Puerto donde se encuentra el servidor de management. Por defecto el valor es 9090.

`-p puerto-local`
Puerto TCP donde escuchará por conexiones entrantes HTTP. Por defecto el valor es 8080.

`-v`
Imprime información sobre la versión y termina.

Para más información del programa, ejecutar dentro de la carpeta raíz

```
$ man ./httpd.8
```

Argumentos de línea de comando para httpdctl

`-h`
Imprime el manual y finaliza.

`-v`
Imprime la versión del programa `./httpdctl` y finaliza.

`-p puerto-proxy`
Puerto UDP donde el servidor PCAMP escucha. Por defecto toma el valor 9090.

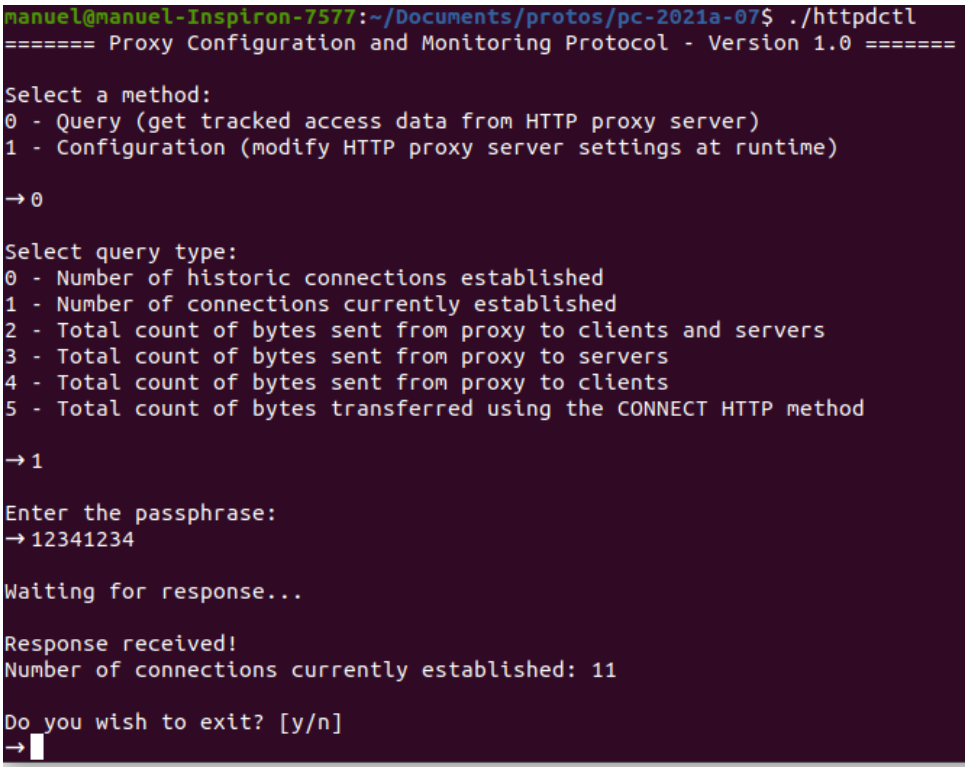
`-l dirección-proxy`

Establece la dirección donde el servidor PCAMP escucha. Por defecto toma el valor 127.0.0.1.

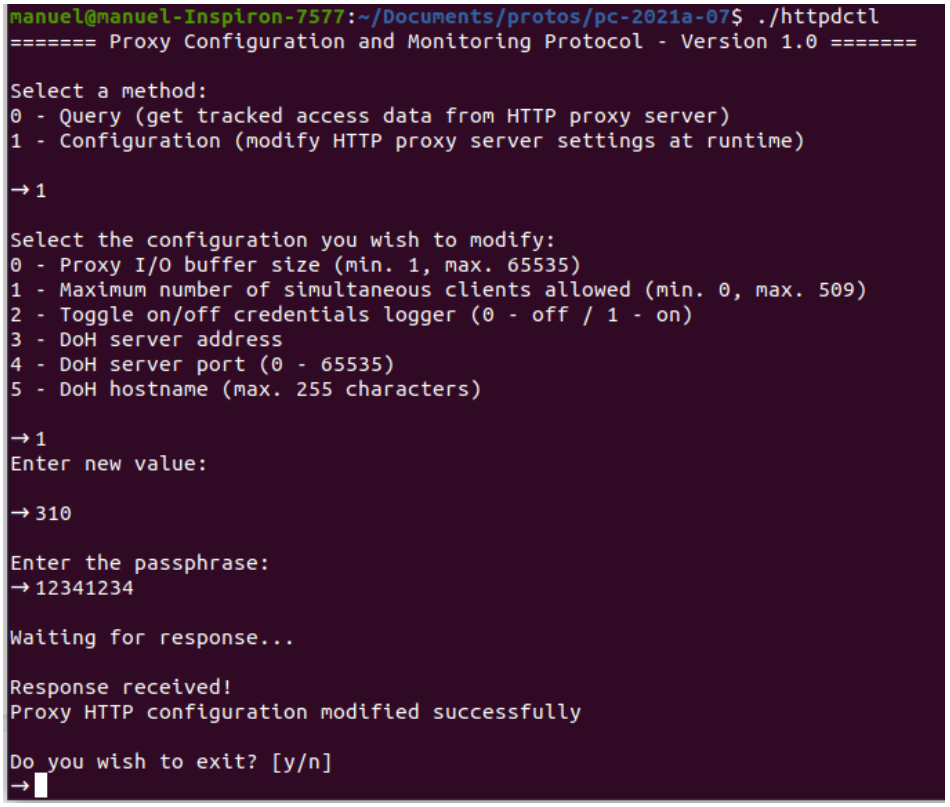
Ejemplos de configuración y monitoreo

Corriendo el ejecutable ./httpdctl, podemos utilizar una interfaz gŕafica para realizar una query de monitoreo o cambio a una configuración.

Ejemplo de monitoreo de total de conexiones actuales:



Ejemplo de configuración del máximo total de clientes simultáneos:



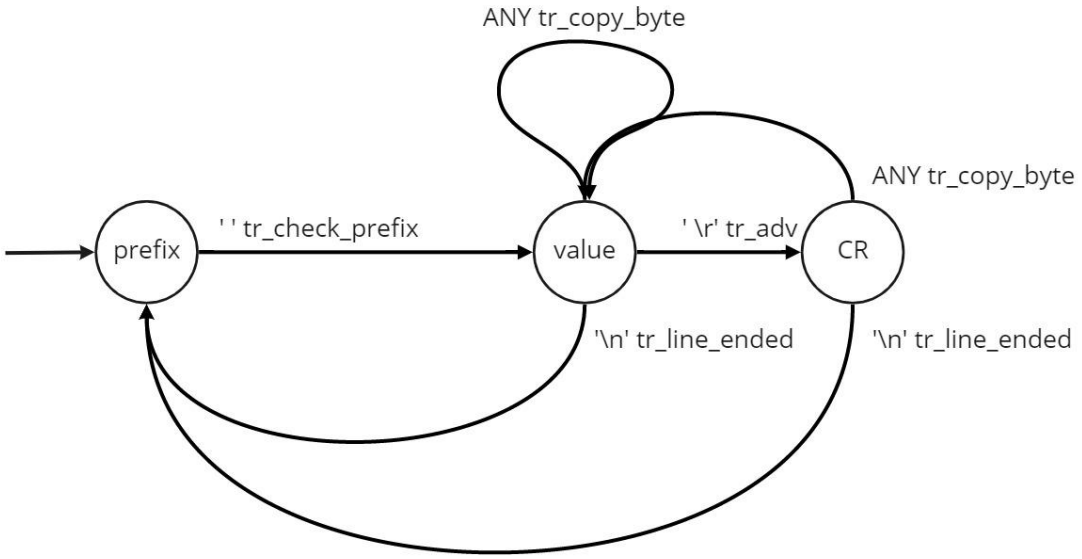
Documento de diseño del proyecto

Grafo del parser HTTP

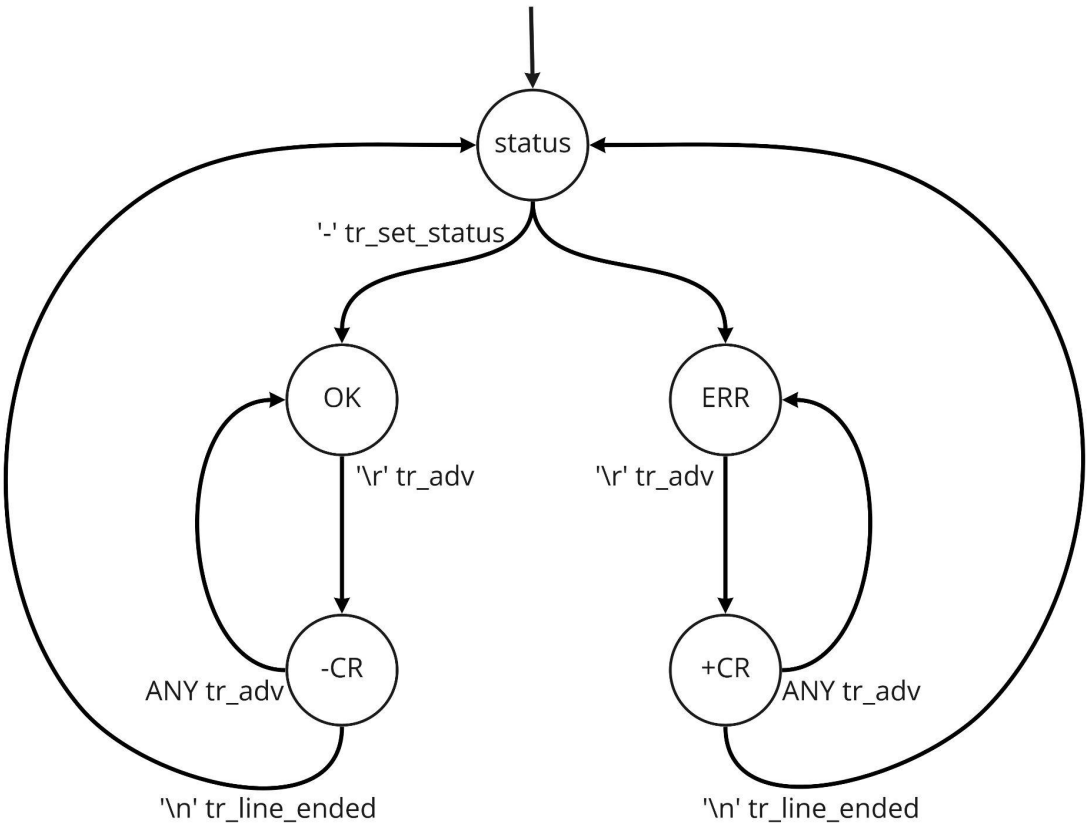
Está adjunto en documento `HTTP_Parser_graph.pdf` en la carpeta raíz del proyecto.

Nota: el grafo sólo muestra los caracteres necesarios para realizar las transiciones, si con el carácter actual no puede viajar a otro nodo el estado pasará a ser de *error*. Se omitió el estado trampa y las funciones ejecutadas en cada transición para simplificar la comprensión.

Grafo del parser POP3 command



Grafo del parser POP3 response



Especificaciones técnicas del protocolo de monitoreo y configuración PCAMP

Consultar el archivo `especificaciones-pcamp.pdf` dentro de la carpeta raíz del proyecto para leer la documentación de diseño y especificaciones del protocolo.

Bibliografía

1. Fielding, R., Reschke, J., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", rfc7230, Junio 2014.
2. Fielding, R., Reschke, J., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", rfc7231, Junio 2014.
3. Fielding, R., Reschke, J., "Hypertext Transfer Protocol (HTTP/1.1): Conditional Requests", rfc7232, Junio 2014.
4. Berners-Lee, T., Fielding, R., Masinter, L., "Uniform Resource Identifier (URI): Generic Syntax", rfc3986, Enero 2005.
5. Myers, J., Rose, M., "Post Office Protocol - Version 3", rfc1939, Mayo 1996.
6. Hinden, R., Carpenter, B., Masinter, L., "Format for Literal IPv6 Addresses in URL's", rfc2732, Diciembre 1999.
7. Mockapetris, P., "Domain Names - Implementation and Specification", rfc1035, Noviembre 1987.
8. Hoffman, P., McManus, P., "DNS Queries over HTTPS (DoH)", rfc8484, Octubre 2018.
9. Hinden, R., Deering, S., "IP Version 6 Addressing Architecture", rfc4291, Febrero 2006.

Fuentes externas de código

1. `base64.h` de William Sherif
    - Repositorio: https://github.com/superwills/NibbleAndAHalf/blob/master/NibbleAndAHalf/base64.h
    - Hash del último commit utilizado en el archivo: `819101372006b66ad88a957b48c627b7145e901c`
    - Modificaciones al código: se cambiaron en las líneas 29 y 32 los "`const static`" por "`static const`" y se agregó un '`\n`' al final del archivo para garantizar la compilación correcta contra esta los flags utilizados en el proyecto.