Crypto-Sockets

Ejercicio N° 1

Objetivos	● Buenas prácticas en programación de Tipos de Datos Abstractos (TDAs) ● Modularización de sistemas ● Correcto uso de recursos (memoria dinámica y archivos) ● Encapsulación y manejo de Sockets
Instancias de Entrega	**Entrega 1:** clase 4 (20/10/2020). **Entrega 2:** clase 6 (03/11/2020).
Temas de Repaso	● Uso de structs y typedef ● Uso de macros y archivos de cabecera ● Funciones para el manejo de Strings en C ● Funciones para el manejo de Sockets
Criterios de Evaluación	● Criterios de ejercicios anteriores ● Cumplimiento de la totalidad del enunciado del ejercicio ● Ausencia de variables globales ● Ausencia de funciones globales salvo los puntos de entrada al sistema (*main*) ● Correcta encapsulación en TDAs y separación en archivos ● Uso de interfaces para acceder a datos contenidos en TDAs ● Empleo de memoria dinámica de forma ordenada y moderada ● Acceso a información de archivos de forma ordenada y moderada

Índice

Introducción

En este ejercicio se deberán desarrollar dos programas: uno emisor (o cliente), y otro receptor (o servidor) de un mensaje cifrado.
El programa emisor deberá leer mensajes por entrada estándar y enviarlos cifrados, mientras que el receptor deberá descifrarlos y mostrarlos por salida estándar.

Descripción

Se deberán implementar los siguientes algoritmos de cifrado de *clave simétrica* [1]:
- Cifrado de César (en realidad vamos a programar su generalización) [2]
- Cifrado de Vigenere [3]
- Rivest Cipher 4 [4]

A continuación, se dará un breve resumen del funcionamiento de cada uno, pero se recomienda buscar en internet para mayor detalle. Tener una tabla ascii a mano.

Cifrado de César

El cifrado de César consiste en desplazar cada caracter de la cadena a cifrar en 3 posiciones. Así, por ejemplo, el caracter `'A'` cifrado se convierte en un caracter `'D'`, mientras que una `'h'` se convierte en una

`'k'`.
Una generalización muy directa de este algoritmo es pensar ese 3 como un valor de clave secreta N, y pasar a desplazar cada caracter en N posiciones. Entonces, si fijamos N=7, la `'A'` se convierte en `'H'`, y la `'h'` en `'o'`.
Otra aclaración importante sobre la implementación: para simplificar, vamos a considerar que cada **char** de entrada es un caracter del mensaje a cifrar (no consideramos encoding), y que el diccionario consta de 256 valores. Así, la letra `'Z'` cifrada con N=1 nos va a devolver el caracter `'['`, y no la letra `'A'`.
En resumen, mirá cada byte como un unsigned, sumalos y haceles módulo 256.

Cifrado de Vigenere

El cifrado de Vigenere es una generalización algo más amplia que la anterior, pero ahora *polialfabética* (la operación de sustitución ahora cambia a lo largo del mensaje).
A diferencia del método anterior, ahora la clave no es un número sino una cadena, y lo que haremos para decidir el valor de la salida es sumar el valor de cada caracter de entrada con el caracter correspondiente de la clave.
Expliquemos esto con un ejemplo (y ya que estamos, despejemos dudas de mapeos): Si el mensaje es "Secret message" y la clave secreta es "SecureKey", tendremos la siguiente suma:

```
Message: |S|e|c|r|e|t| |m|e|s|s|a|g|e|
Key:     |S|e|c|u|r|e|K|e|y|
```

En este punto podrían surgir algunas dudas, como cuánto vale una 'S', o qué pasa con los caracteres del mensaje de entrada que no tienen un valor de la key abajo. Respecto al valor de las letras, vamos a tomar el valor que nos indique la tabla ascii. Entonces reemplazando con valores decimales, la tabla de arriba queda:

```
Message: |083|101|099|114|101|116|032|109|101|115|115|097|103|101|
Key:     |083|101|099|117|114|101|075|101|121|
```

Y respecto de qué pasa con los valores del mensaje que no tienen su índice correspondiente en la clave, se deben volver a usar los primeros valores de la misma. Entonces la suma es:

```
Message: |083|101|099|114|101|116|032|109|101|115|115|097|103|101|
Key:     |083|101|099|117|114|101|075|101|121|083|101|099|117|114|
Encoded: |166|202|198|231|215|217|107|210|222|198|216|196|220|215|
```

¡Advertencia! Tanto el cifrado de Vigenere como el de César dependen del alfabeto que utilicemos, en muchas páginas de internet van a encontrar herramientas que consideran el alfabeto de la A a la Z, considerando o no la letra Ñ, y varias consideraciones por el estilo. Nosotros usaremos su valor ASCII, así el alfabeto es de 256 valores y es más sencillo manejarlo en C. Tengan esto en mente si van a corroborar el correcto funcionamiento de su algoritmo contrastando sus salidas con alguna de estas herramientas.

Rivest Cipher 4

De los algoritmos criptográficos con uso productivo este es uno de los más simples, pero es en varios niveles más complejo que los otros dos que se deben implementar en este TP. Lleva este nombre en honor a Ronald Rivest, que también es uno de los autores del libro "Introduction to Algorithms" conocido en nuestra fiuba como "el Cormen" porque tiene un apellido alfabéticamente conveniente, es una de las bases del protocolo deprecado WEP, y también es usado en TLS.

RC4 genera un stream “infinito” a partir de una clave secreta, y a cada caracter de entrada le aplica una operación XOR con el siguiente byte en el stream.
Para ver detalles sobre la generación del stream y sobre cómo aplicar la operación XOR, ver la página de Wikipedia https://es.wikipedia.org/wiki/RC4, de la cual está permitido sacar ideas para la implementación del algoritmo. Por supuesto, se debe mejorar la calidad del código ya que el que está en Wikipedia utiliza variables globales y pésimos nombres, lo cual llevaría a la desaprobación del ejercicio.

Formato de Línea de Comandos

Todos los algoritmos de cifrado requeridos son de clave simétrica, por lo que ambos procesos necesitarán conocer la clave secreta para funcionar correctamente.

Además, como se deberán implementar varios algoritmos, también se recibirá por línea de comandos el método a utilizar.

El cliente se ejecutará usando el siguiente formato:

```
./client <server-host> <server-port> --method=<method> --key=<key>
```

Y el servidor con el siguiente:

```
./server <server-port> --method=<method> --key=<key>
```

El argumento “method” podrá recibir los siguientes valores:

- cesar
- vigenere
- rc4

Los argumentos siempre se recibirán en ese orden (por ejemplo, nunca va a aparecer primero la key y después el method).

Estructuración del Código

El código entregable deberá ser subido en un .zip al Sercom, con todos los fuentes en la raíz del mismo, y con dos funciones main (una para cada proceso). Para organizar, el Makefile toma todos los archivos que comienzan con “`server_`” y los que comienzan con “`common_`” para construir el server; y todos los archivos que comienzan con “`client_`” junto con los que comienzan con “`common_`” para construir el client.
Tener esto en cuenta antes de arrancar.

Protocolo de Comunicación

Los datos que “viajarán” por la red serán solamente en sentido emisor -> receptor (client -> server) y serán ni más ni menos que el mensaje cifrado.
El receptor se va a enterar que el emisor terminó su envío, porque el emisor cerrará el socket una vez que haya terminado el mensaje (recordar/rever cómo se entera un proceso de que el otro extremo le cerró la conexión).

Códigos de Retorno

Ambos procesos retornarán 0 en cualquier caso.

Entrada y Salida Estándar

El mensaje plano será recibido por entrada estándar en el cliente, y es la salida esperada por salida estándar en el servidor. El resto de los flujos estándar se asumirán vacíos (sercom no va a validar salida estándar, así que se pueden usar para logging propio).

Ejemplos de Ejecución

A continuación veremos algunos ejemplos de ejecución mostrando entradas y salidas esperadas.

Ejemplo 1: RC4, Pan con queso

Empecemos por el método más difícil.
Sea el siguiente archivo, llamado "__client_stdin__":

```
Pan
```

Ejecutamos el servidor en el puerto 8080, configurándolo para que use RC4:

```
./server 8080 --method=rc4 --key=queso
```

Luego ejecutamos el cliente, configurándolo de una manera compatible con ese servidor, y redirigiendo la entrada estándar:

```
./client 127.0.0.1 8080 --method=rc4 --key=queso < __client_stdin__
```

El cliente deberá conectarse al servidor y cifrar los datos de entrada, obteniendo los bytes:

```
69|ca|e6
```

Y luego enviarlos a través del socket. A continuación, cerrará el socket.
El servidor recibirá el mensaje cifrado y lo descifrará, obteniendo el mensaje original. Debe imprimir por salida estándar lo obtenido (como es de suponer, debe coincidir con la entrada estándar del cliente).

```
Pan
```

Luego de realizar lo descripto arriba, ambos procesos deben terminar ordenadamente, liberando los recursos usados durante la ejecución.

Ejemplo 2: César

Usando el mismo archivo de entrada, veamos qué sucede con el método de César.

Sea el archivo "__client_stdin__":

```
Pan
```

Ejecutamos el servidor, pero ahora lo configuramos para que use el cifrado de César. Notar que en este método la clave es un número y no un string:

```
./server 8080 --method=cesar --key=5
```

Análogamente, el cliente lo ejecutamos así:

```
./client 127.0.0.1 8080 --method=cesar --key=5 < __client_stdin__
```

El cliente va a cifrar los datos (shifteando 5 posiciones a cada char), entonces EN HEXA los bytes quedan:

```
55|66|73
```

Los envía al servidor, cierra el socket, el servidor los descifra, y los imprime descifrados en salida estándar:

```
Pan
```

Nota: Los valores están puestos con cuidado, pero puede que alguno esté mal. ¡Si encontrás algún error reportalo y lo cambiamos!

Restricciones

La siguiente es una lista de restricciones técnicas exigidas por el cliente:

1. El sistema debe desarrollarse en ISO C (C99) / ISO C++11.
2. Está prohibido el uso de variables y funciones globales.
3. Todo socket utilizado en este TP debe ser **bloqueante** (es el comportamiento por defecto).
4. Las funciones deben ser **cortas** (idealmente de menos de 10 líneas) y el código **claro**.
5. El archivo de entrada se debe leer en chunks de 64 Bytes, usando el stack.

Recomendaciones

Los siguientes lineamientos son claves para acelerar el proceso de desarrollo sin pérdida de calidad:

1. Tener siempre a mano las páginas de manual, y al abrirlas leerlas con mucha atención. Sobre todo para las funciones que no hayas usado nunca: no cuesta nada escribir 'man send' o 'man recv' en la consola y leer bien qué dice cada una sobre los valores del último parámetro o sobre los valores de retorno.
2. ¡Programar por bloques! En este TP tenés que programar tres cifradores, que no tienen motivos para saber nada sobre sockets. ¡Si los programás así, podés hasta hacerles tests fáciles antes de entregar!. Teniendo cada bloque, es fácil armar un bloque de más alto nivel que los use.
3. Una vez que tengas los cifradores preguntate lo siguiente antes de seguir con los sockets y hacé un diagrama (si te sale prolijo lo podés entregar en el informe, ya que es obligatorio tener al menos un diagrama): ¿Debería el TDA Socket saber sobre algún cifrador? ¿Debería, por el contrario, saber algún cifrador sobre TCP? ¿Y quién lee de entrada estándar?
4. Escribí código claro, sin saltos en niveles de abstracción, y que puedas leer entendiendo qué está

pasando. Si editás el código “hasta que funciona” y cuando funcionó lo dejás así, buscá la explicación de por qué anduvo. En el informe también escribí claro, en impersonal, y evitando opiniones.

5. No te compliques la vida con diseños complejos. Cuanto más fácil mejor.
6. Empezá leyendo el archivo de a un byte, y después agrandá eso para que lea el tamaño especificado en las restricciones.

Referencias

[1] Criptografía de Clave Simétrica: https://es.wikipedia.org/wiki/Criptograf%C3%ADa_sim%C3%A9trica
[2] Cifrado de César: https://es.wikipedia.org/wiki/Cifrado_C%C3%A9sar
[3] Cifrado de Vigenere: https://es.wikipedia.org/wiki/Cifrado_de_Vigen%C3%A8re
[4] Rivest Cipher 4: https://es.wikipedia.org/wiki/RC4