CASO 3 – PARTE 2 – CANALES SEGUROS

Universidad de los Andes Departamento de Ingeniería de Sistemas y Computación
ISIS 2203 Infraestructura Computacional
202220

Santiago Diaz Moreno 201912247
Nicolás Klopstock Triana 202021352
Julián Camilo Mora Valbuena 20201274

Contenido

1. Contexto del problema

Para resolver el caso 3, se modeló un programa que simula las comunicaciones entre clientes y el servidor de una compañía que ofrece consultas en línea, garantizando la confidencialidad e integridad de la información. Con base en lo anterior, se propuso una arquitectura cliente-servidor, en la cual se comunican mediante sockets. Adicionalmente, para asegurar que se cumplieran las métricas de seguridad, se utilizaron los algoritmos AES, CBC y PKCS5Padding, de cifrado simétrico, RSA, de cifrado asimétrico y HMACSHA256 para código criptográfico.

2. Descripción de la solución

La solución propuesta está basada en el protocolo descrito en el enunciado del caso. Este sigue 13 pasos de interacción entre el servidor web y el cliente. Lo primero es "comenzar" la comunicación con un mensaje del cliente "SECURE INIT". Cuando el servidor recibe este mensaje, manda cuarto datos, tres de los cuales se usan para generar el valor que se usará como semilla para la llave simétrica. El dato restante de los cuatro mencionados es la firma del servidor, para ser verificada por el cliente. Si todo está correcto con la firma del servidor, el cliente envía un mensaje de "OK", de lo contrario, uno de "ERROR". Con este mensaje de confirmación, el cliente también envía su valor de y calculado. Para este momento, ambas partes tienen el valor y calculado por el otro, entonces pueden calcular el valor de z para generar la llave simétrica. Además, con el mismo valor z también generan la llave simétrica para el valor de integridad y un vector de inicialización de bits random. Con estos valores, el cliente procede a generar un entero random y lo cifra, y genera su código de integridad. Todo esto, junto al vector de inicialización, lo envía al servidor (por su parte, el servidor hace lo mismo). Una vez enviados los tres datos mencionados, el servidor manda un mensaje de "OK" si sus verificaciones del cifrado y de integridad son positivas o "ERROR" si son negativas. Luego, internamente en el servidor, se descifra el mensaje enviado, se convierte a entero y se le suma uno (+ 1). Este nuevo mensaje se cifra y se le envía al cliente junto con el código de verificación de integridad del nuevo mensaje y un nuevo vector de inicialización. Con esto, el cliente descifra el mensaje, verifica que, en efecto, es el entero original más uno (+ 1) y que el código

de integridad sea el mismo. Si todo sale bien, el cliente manda un mensaje de "OK" y acaba el proceso; de lo contrario, manda un mensaje de "ERROR".

Es importante aclarar que, en cada ejecución del servidor, este decide de forma aleatoria, uno de tres escenarios posibles. El primero (Test 0) "sale bien" si la firma es la incorrecta y el servidor recibe el mensaje de "ERROR". El segundo (Test 1) "sale bien" si la verificación de integridad luego de que el cliente ya haya recibido el mensaje original más uno cifrado es de resultado negativo. Por último, está el tercero (Test 2), que sale bien si cada paso del protocolo no tiene ninguna falla y el mensaje final es el esperado.

3. Preguntas

3.1. En el protocolo descrito el cliente conoce la llave pública del servidor (K_w+). ¿Cuál es la manera común de enviar estas llaves para comunicaciones con servidores web?

Para enviar llaves a través de medios inseguros se utilizan algoritmos como Diffie-Hellman para el cálculo de llaves privadas en común.
Otra manera de hacer la entrega de llaves es de manera física, entregarse las llaves a través de una memoria usb o algún paquete sellado permite asegurarse que el destinatario sea la persona que queremos que reciba la llave, ya además nos cercioramos de que la información sea la correcta y que no se modificara durante el proceso de envío.

3.2. El protocolo Diffie-Hellman garantiza "Forward Secrecy", explique en qué consiste esta garantía.

"Forward Secrecy" es un término que se refiere al intercambio seguro de llaves privadas en las comunicaciones cliente-servidor. Este término es utilizado para describir el adecuado resultado de dichos intercambios, en tanto permanecen completamente secretos para un tercero en el medio, pero son congruentes entre cliente y servidor.

El algoritmo Diffie-Hellman, consiste en el cálculo de llaves en común por medio de operaciones matemáticas.
Para ello, los dos usuarios (A y B) calculan una llave pública con un número primo *p*, una base *g*, menor que *p*, y un número propio *x*, menor que *p-1*. La llave se calcula de la siguiente manera:

$$y = g^x mod p$$

Ambos usuarios, envían su llave pública *y* por el medio inseguro, y, utilizando sus números *x* propios, pueden generar una llave privada en común *z* de la siguiente manera:

$$z = y^x mod p$$

En conclusión, algoritmo Diffie-Hellman se basa en procesos matemáticos, computacionalmente difíciles de descifrar sin todos los datos, con el objetivo de generar dos llaves privadas en común, sin la necesidad de compartir toda la información a través de medios no seguros. Como el resultado de dichas operaciones e intercambios resulta en que cada usuario genera, por su cuenta, una llave privada en común, que no puede ser calculada por un agente externo, se afirmar que Diffie-Hellman cumple con "Forward Secrecy".

4. Escenarios de prueba

Se realizaron pruebas para tres escenarios distintos, en los que se alteraba el número de clientes delegados entre 4, 16 y 32. Para cada escenario se tomaron los tiempos en tres diferentes ocasiones y se registraron los promedios de todos en la tabla 1.

5. Resultados de los escenarios de prueba

Paso de protocolo / # delegados	**4**	**16**	**32**
Cifrar consulta	1158035	6940625	9768300
Generar código de autenticación	1289400	3746200	4423000
Verificación de la firma	8087300	23894550	63630600
G^y	4716900	10576100	12068800

Tabla 1: Resultados ponderados de los escenarios de prueba

(todos los tiempos recopilados están en nanosegundos.)

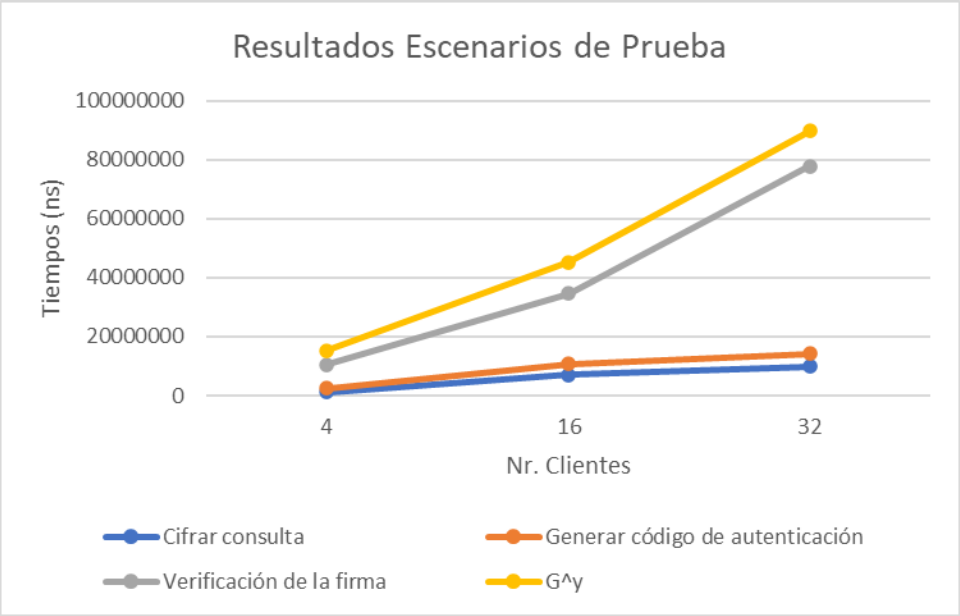

Figura 1: Gráfica de los resultados para los escenarios de prueba

6. Conclusiones de los datos

- Las proporciones de los tiempos se deberían mantener, sin importar el número concurrente de clientes.
- Es muy importante sacar resultados ponderados, ya que dependen mucho de cuántos threads entraron a qué test del servidor.
- A medida que aumenta el número de clientes, los tiempos deben, en promedio, aumentar; y las proporciones de los mismos se tienen que mantener.
- Otra razón por la que los tiempos deben ser ponderados, es porque el número aleatorio que usamos como consulta cambia para cada prueba. Esto afecta a gran escala los tiempos mencionados de cifrado, generación de código de autenticación y su verificación.
- Tiene sentido que las acciones de *"Cifrar consulta"* y *"Generar código de autenticación"* sean las que menos tiempo tardan, ya que usan llaves simétricas. Esto hace que sus procesos sean más rápidos que, por ejemplo, *"Verificación de la firma"*, una acción que usa una llave pública, lo que hace que sea más tardado.

7. Estimaciones de resultados de procesador

$$Velocidad\ de\ procesador\ =\ n\acute{u}cleos\ \cdot\ reloj\ base$$

$$Velocidad\ de\ procesador\ =\ 4\ n\acute{u}cleos\ \cdot\ 2{,}8\ GHz$$

$$Velocidad\ de\ procesador\ =\ 11{,}2\ GHz$$

$$1158035\ ns\ \rightarrow\ 3\ cifrados$$

$$0{,}001158035\ s\ \rightarrow\ 3\ cifrados$$

$$\frac{(0{,}001158035\ s\ \cdot\ 1)}{3}\ =\ 0{,}000386012\ s$$

$$0{,}000386012\ s\ \rightarrow\ 1\ cifrado$$

$$\frac{1}{0{,}000386012\ s} = 2590{,}6\ cifrados$$

$$La\ máquina\ puede\ cifrar\ 2590{,}6\ consultas\ en\ 1\ segundo.$$

$$1289400\ ns\ \rightarrow\ 3\ cifrados$$

$$0{,}001289400\ s\ \rightarrow\ 3\ cifrados$$

$$\frac{(0{,}001289400\ s\ \cdot\ 1)}{3} = 0{,}000429800\ s$$

$$0{,}000429800\ s\ \rightarrow\ 1\ cifrado$$

$$\frac{1}{0{,}000429800\ s} = 2326{,}7\ cifrados$$

La máquina puede cifrar 2326,7 *consultas en* 1 *segundo.*

$$8087300\ ns \rightarrow 3\ verificados$$

$$0.00238495\ s \rightarrow 3\ verificados$$

$$\left(\frac{0.00238495\ s \cdot 1}{3}\right) = 0{,}000794983\ s$$

$$0{,}000794983\ s \rightarrow 1\ verificado$$

$$\left(\frac{1}{0{,}000794983\ s}\right) = 1257{,}8\ verificaciones$$

La maquina puede realizar 1257,8 *verificaciones de firma en* 1 *segundo.*

8. Organización de los archivos en el zip

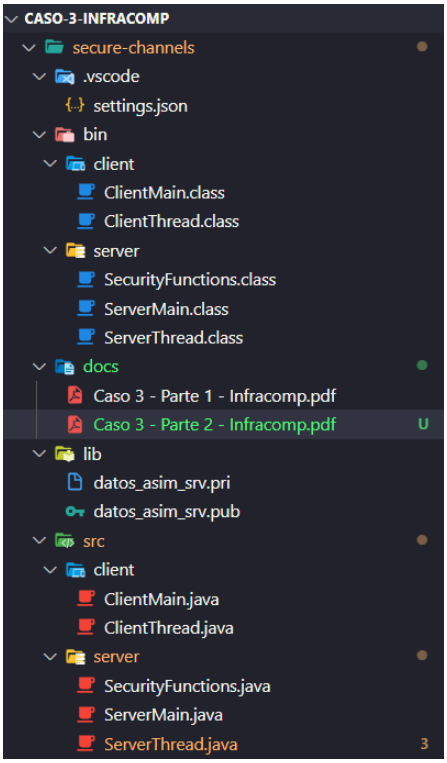

Figura 3: Organización del espacio de trabajo en Visual Studio Code

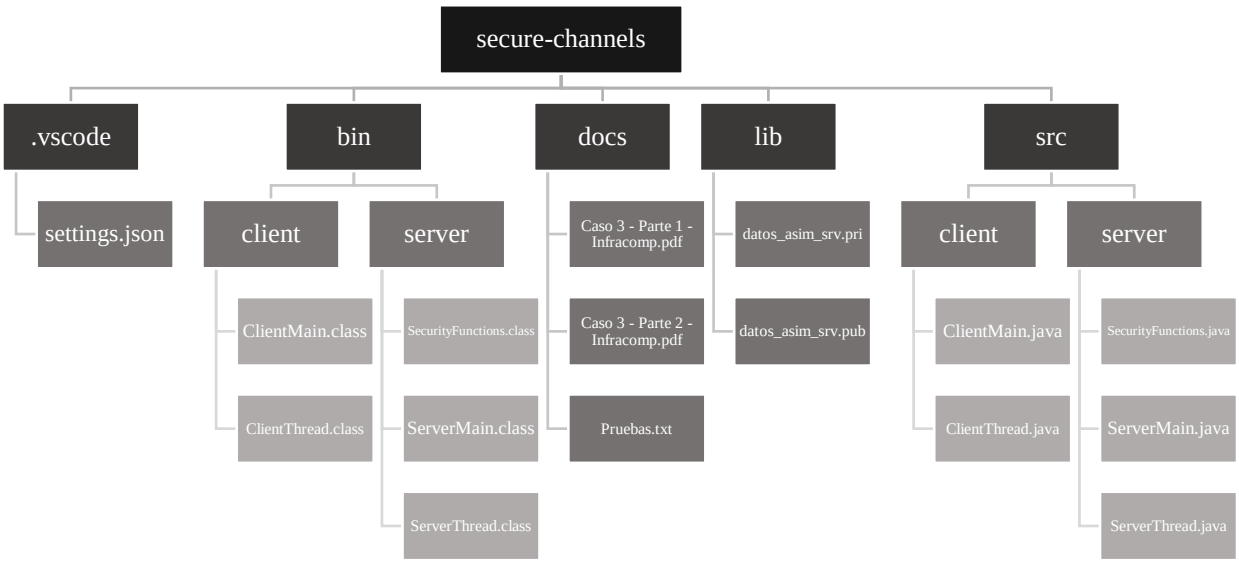

Figura 4: Estructura jerárquica del proyecto

Así como se puede observar en las figuras 3 y 4, dentro del zip tenemos un directorio principal llamado secure-channels, dentro del cual se encuentran todos los archivos correspondientes al caso 3, distribuidos en subdirectorios. En primer lugar, tenemos los subdirectorios .vscode, bin, docs y lib, en los cuales se almacena la configuración del espacio de trabajo, el código compilado, los informes para ambas partes del caso y las llaves privadas y públicas, respectivamente. Cabe destacar, que, para simplificar la toma de los tiempos en los resultados, implementamos métodos que escribían los tiempos en archivos de prueba.txt, los cuales almacenamos en el subdirectorio docs. Asimismo, dentro de secure-channels se encuentra el subdirectorio src, donde a su vez, se guardan dos subdirectorios adicionales, los cuales contienen el código fuente del caso, repartidos en dos secciones: cliente y servidor.

Dentro de cliente, se hallan dos archivos `ClientMain.java` y `ClientThread.java`. En el primero, se encuentra el código para inicializar los clientes, los cuales serán manejados como Threads de java. A estos, se le asigna la conexión a un servidor, mediante sockets y un puerto determinado. En el segundo archivo, se encuentra el código donde se hacen las operaciones de seguridad para cada cliente, como la implementación del algoritmo Diffie-Hellman para intercambio de llaves, así como los cifrados y descifrados (en resumen, todo el lado del cliente del protocolo descrito).

Por el lado del servidor, tenemos tres archivos, dos de los cuales funcionan similar a los explicados anteriormente, `ServerMain.java` y `ServerThread.java`. En el primero, se crea el servidor, un delegado por cliente para atender cada conexión, que a su vez va a ser manejado con Threads de Java. Además, se establecen las conexiones con clientes mediante sockets y puertos. En el segundo archivo, se encuentran las funciones de los Threads para el servidor, en el cual se llevan a cabo los procesos relacionados con criptografía, como el cálculo de llaves a través de las operaciones matemáticas descritas en el algoritmo Diffie-Hellman, así como cifrados y descifrados para validar la integridad y confidencialidad de los archivos. Finalmente, el tercer archivo dentro del subdirectorio server se llama `SecurityFunctions.java`, en este

se guardan las instancias de los algoritmos utilizados para cifrado simétrico (AES, CBC Y PKCS5Padding), asimétrico (RSA) y código criptográfico de Hash (HMACSHA256).

9. Instrucciones para correr el programa

1. Abrir la carpeta “secure-channels”
2. Correr el archivo `ServerMain.java`.
3. Correr el archivo `ClientMain.java`.
4. Ingresar el número de clientes que se quiera.

10. Referencias

(2022). *Criptografía* [Diapositivas de PowerPoint]. Departamento de Ingeniería de Sistemas y Computación, Universidad de los Andes. https://bloqueneon.uniandes.edu.co//content/enforced/134477-202220_ISIS2203_03/M3C2-Criptografi%CC%81a.pdf?_&d2lSessionVal=AozfGvOUvhRwNKPWtiThiYtEf&ou=134477

F5 DevCentral. (2017, 25 de abril*). Perfect Forward Secrecy* [Video]. YouTube. https://youtu.be/IkM3R-KDu44