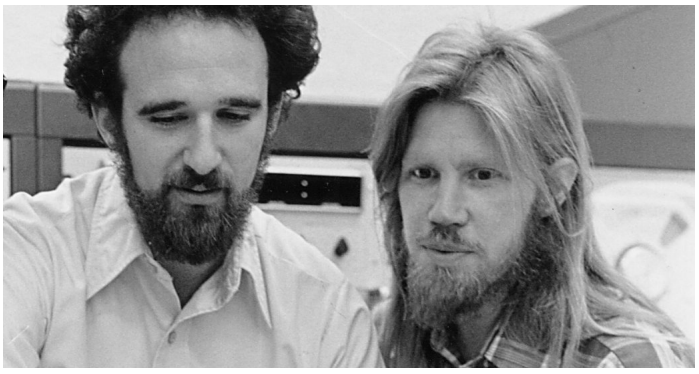
Universidad Autónoma de Madrid

Grado de Ingeniería Informática

Redes de Comunicaciones II

Protocolo de establecimiento de claves Diffie-Hellman

Un modelo robusto de cifrado para mensajería instantánea



Autor: Alejandro Santorum

Profesor: Óscar Delgado

Año académico 2018-2019

Indice

1

Introducción

El trabajo expuesto en la siguiente memoria recoge el proceso de investigación e implementación de un servidor que simula un **servicio de mensajería instantánea** entre sus posibles clientes. El objetivo principal es establecer contacto con el **protocolo de establecimiento de claves Diffie-Hellman**, por lo que los mensajes circularán por la red cifrados utilizando las claves acordadas con dicho protocolo.

Además de asegurar la confidencialidad de los mensajes, nos hemos interesado por la integridad de los mismos y la autentificación de los interlocutores de los mensajes, por lo que se expondrá un modelo criptográfico híbrido, utilizando diferentes algoritmos.

Por último, la comunicación entre los clientes y el servidor estará implementada mediante ciertos comandos que se expondrán más adelante.

2

Protocolo Diffie-Hellman

Se comenzará exponiendo el tema central de este trabajo: el **protocolo criptográfico Diffie-Hellman**. Este es un protocolo de establecimiento de claves ideado por los criptógrafos **Bailey Whitfield Diffie** y **Martin Edward Hellman**.

El protocolo está ideado para acordar una clave de cifrado simétrico entre partes que no han tenido contacto previo, y se intentan comunicar en un medio potencialmente inseguro.

2.1 Descripción

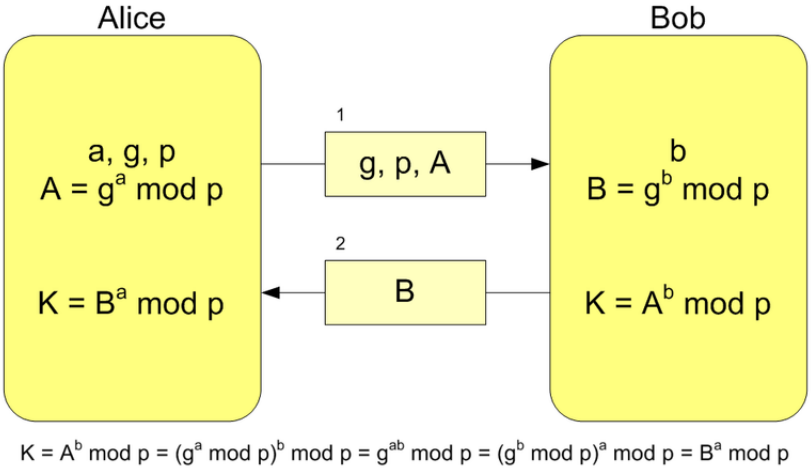
El protocolo está diseñado para que dos interlocutores puedan una clave compartida a través de un canal inseguro, por lo que puede haber otros agentes que deseen interceptar dicha clave para realizar acciones maliciosas.

La seguridad del algoritmo radica en la extrema dificultad, con los recursos computacionales actuales, de calcular logaritmos discretos en un cuerpo finito, es decir, se realiza una operación potencialmente no invertible para que sea imposible (repitimos, en la actualidad) de obtener el secreto a partir de las partes públicas.

2.2 Detalles del algoritmo

Supongamos dos interlocutores, típicamente en la literatura criptográfica llamados Alicia y Bernardo. Entre los dos (de forma pública) acuerdan un un primo p, cuanto más grande más seguro, y un generador g tal que $g \in \mathbf{Z}_p^*$, es decir, un elemento menor y coprimo con p.
Ahora Alice escoge aleatoriamente a tal que $a \in \mathbf{Z}_{p-1}$ y calcula $A = g^a \ (mod\, p)$ y envía A a Bernardo.
Mientras tanto, Bernardo escoge también $b \in \mathbf{Z}_{p-1}$ al azar y calcula $B = g^b \ (mod\, p)$, enviando el resultado a Alicia.

Una vez que Alicia recibe B y Bernardo A, pueden ambos calcular $K = A^b \,(mod\, p) = B^a \,(mod\, p) = g^{ab} \,(mod\, p)$. Ahora ambas partes (y únicamente ellas) poseen el valor de la constante K y pueden utilizarla como clave compartida para pasarse mensajes secretos utilizando un algoritmo criptográfico de clave simétrica.



2.3 Generalizaciones

2.3.1 N partes

Para extender el mecanismo a grupos de tamaño mayor, supongamos de tamaño N, se deben seguir dos reglas básicas:
· Comenzamos conociendo el valor g (generador) y p (primo grande) de la misma forma que hacíamos para $n = 2$. Ahora la clave secreta K se obtiene elevando g al exponente secreto de cada parte en cualquier orden.
· Cualquier valor intermedio (habiendo aplicado hasta $N - 1$ exponenciaciones modulares) será revelado públicamente. Así, cada parte podrá obtener finalmente el valor K elevando a su exponente privado.

Un ejemplo básico para $N = 3$ es el siguiente:
1 - Las partes (Alicia, Bernardo y Carolina) se ponen de acuerdo en los parámetros del algoritmo g y p.
2 - Las partes generan sus propios exponentes privados llamados a, b y c respectivamente.
3 - Alica calcula $g^a \,(mod\, p)$ y se lo envía a Bernardo.
4 - Bernardo calcula $(g^a)^b = g^{ab}$ y se lo envía a Carolina.
5 - Carolina calcula $(g^{ab})^c = g^{abc}$ y lo usa como clave secreta K.

6 - Bernardo calcula g^b y se lo envía a Carolina.
7 - Carolina calcula $(g^b)^c = g^{bc}$ y se lo envía a Alicia.
8 - Alicia calcula $(g^{bc})^a = g^{abc}$ y lo usa como clave secreta K.
9 - Carolina calcula g^c y se lo envía a Alicia.
10 - Alicia calcula $(g^c)^a = g^{ac}$ y se lo envía a Bernardo.
11 - Bernardo calcula $(g^{ac})^b = g^{abc}$ y lo usa como clave secreta K.

2.3.2 Cambio de grupo

Podemos utilizar el mismo protocolo pero cambiando el grupo Z_p^* por otros que cumplan las condiciones necesarias para poder aplicar el algoritmo. La búsqueda generalizada de estos grupos es conocido como el **problema generalizado de Diffie-Hellman** (GDHP por sus siglas en inglés).

Algunos ejemplos de variantes de este protocolo son:
· `Curvas Elípticas Diffie-Hellman (ECDH)`: esta variante utiliza curvas elípticas en lugar de grupos multiplicativos de enteros módulo p.
· `Intercambio de claves Diffie-Hellman con isogenias supersingulares (SIDH)`: variante diseñada para ser segura contra ordenadores cuánticos.

3

Modelo criptográfico

El objetivo de esta sección es detallar el modelo criptográfico implementado, que intenta asegurar la **confidencialidad**, **integridad** y **autenticidad** de los mensajes que se intercambien los clientes y el servidor de mensajería.

En primer lugar se realiza el intercambio de claves siguiendo el protocolo Diffie-Hellman para obtener así la clave compartida `K`, que la utilizaremos como clave en un algoritmo de cifrado simétrico.

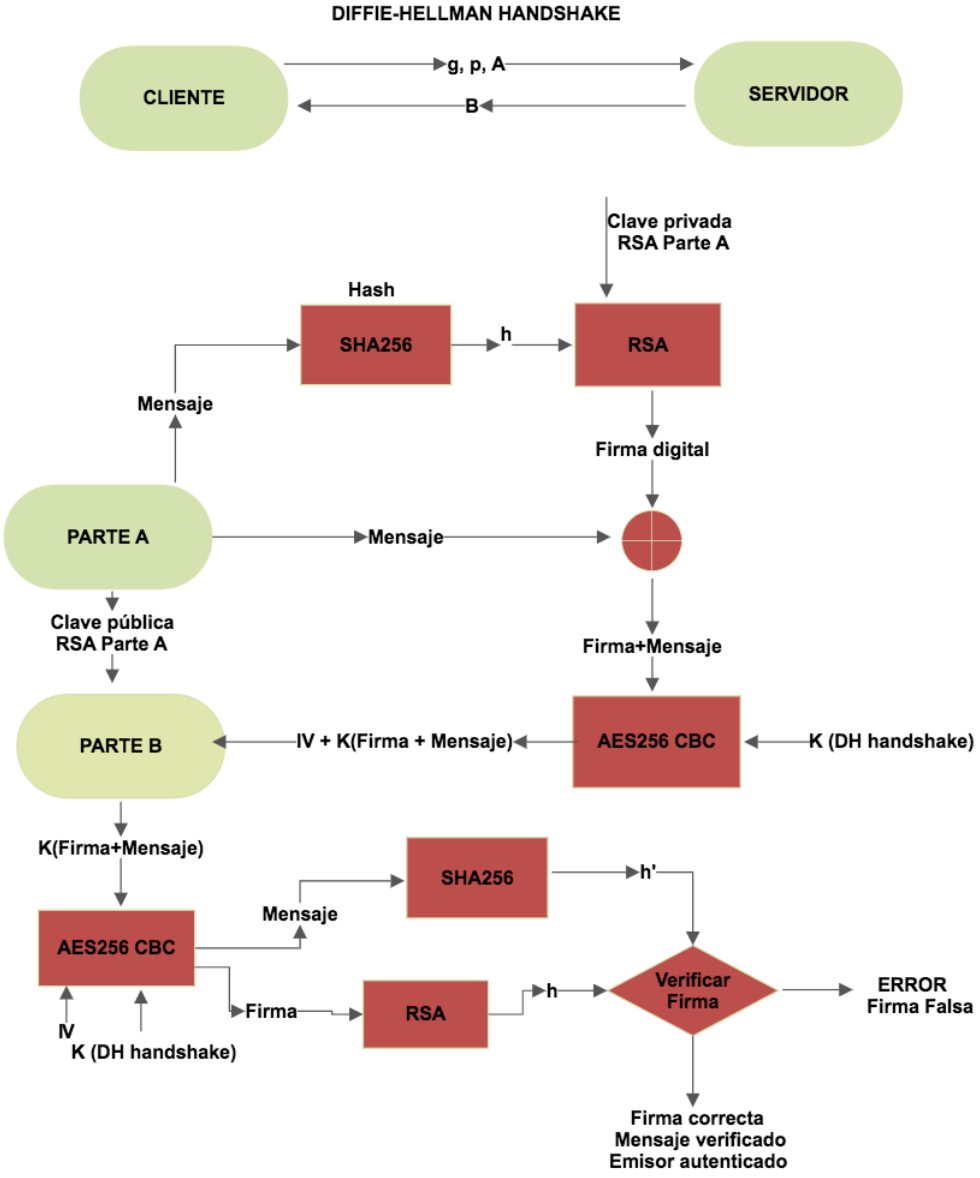
Para poder evitar en todo momento el ataque *Man in the middle* y que un delincuente pueda interponerse entre los mensajes del servidor y un cliente, utilizaremos el algoritmo asimétrico **RSA** junto con el hash **SHA256** para desarrollar una firma digital que asegure la **autenticidad**, tanto del servidor como del cliente. Además, este mismo proceso sirve para asegurar la **integridad** del mensaje, ya que se firmaría el *hash* del mensaje a intercambiar.

Por otro lado, como el mensaje puede ser de una longitud de tamaño considerable, utilizaremos el algoritmo criptográfico simétrico **AES256** con modo *CBC* con vector de inicialización de 16 bytes para cifrar el mensaje, siendo uno de los algoritmos simétricos más robustos de la actualidad y asegurando la **confidencialidad** del mismo. El algoritmo utilizará como clave simétrica la constante K intercambiada previamente.

Finalmente se enviará el mensaje al receptorcon el siguiente formato:
$IV + K_E^{DH}(K_{priv}^{RSA}(H(m)) + m)$, donde IV = vector de inicialización, K_E^{DH} es la clave compartida, K_{priv}^{RSA} es la clave privada del emisor y $H(.)$ es el hash del mensaje. Aquí la + representa la concatenación.

Este proceso más el de descrifrado + verificación es recogido en el siguiente diagrama.

Modelo Criptográfico



4

Servidor Termail

Es la hora de trasladar a la práctica toda la teoría expuesta anteriormente. Para ello hemos implementado en el lenguaje *Python* dos programas, uno con toda la funcionalidad del cliente y otro que establece el servicio de mensajería, el servidor **Termail** (nombre inspirado en la unión de email y terminal).

Suponemos que el servidor ya se encuentra corriendo en una IP pública accesible y un puerto en concreto para poder conectarse a él. Podemos ejecutar el software cliente, el cual primero nos preguntará que acción queremos tomar: `Registrarse`, `Iniciar sesión` o `Salir`.
Para registrarnos el programa nos preguntará por nuestro nombre de usuario (único entre todos los clientes) y una contraseña, la cual tendremos que repetir para evitar posibles confusiones. Una vez introducida esta información, el cliente se conectará a un *socket* del servidor, pidiendo su clave pública RSA y realizando el comentado intercambio de claves Diffie-Hellman. De todo esto no es consciente el usuario, que se limita a esperar unos instantes hasta obtener respuesta.
El comando mandado al servidor es ”REGISTER <nombre> <contraseña> <clave pública RSA cliente>”, que será previamente cifrado y firmado. El servidor descifrará y comprobará la firma y procederá a completar el registro, enviando un comando de respuesta al cliente cifrado, que este también deberá descifrar y verificar, mostrando el resultado al usuario por pantalla.

```
SantorumPC:Termail santorum$ python3 termail_client.py
Select one option:
[1] Register
[2] Sign in
[3] Exit
1
Introduce nickname to be registered: Alice
Introduce password: alice_pass
Reintroduce password again: alice_pass
Registration of user 'Alice' completed
```

El procedimiento seguido para iniciar sesión es bastante similar. El programa nos pedirá en un primer momento nuestro nombre usuario y nuestra contraseña. Ahora el software del cliente pedirá automáticamente la clave pública RSA del usuario con el nombre introducido y establecerá de nuevo una clave compartida cliente-servidor utilizando el protocolo Diffie-Hellman. Una vez obtenida todos estos parámetros necesarios, se mandará al servidor el comando "SIGN_IN <nombre> <contraseña>" cifrado y firmado. El servidor, de nuevo, desencriptará y verificará el mensaje, procediendo a autentificar el inicio de sesión, mandando una respuesta cifrada al cliente.

```
SantorumPC:Termail santorum$ python3 termail_client.py
Select one option:
[1] Register
[2] Sign in
[3] Exit
2
Introduce nickname: Alice
Introduce password: alice_pass
Sign in as 'Alice' successfully
```

Ahora, una vez hemos iniciado sesión, podremos ver la lista de usuarios registrados, comprobar la lista de nuestros mensajes recibidos, leer un mensaje recibido en concreto, enviar un mensaje a un usuario dado Y cerrar la sesión. Todo esto podemos recordarlo en cualquier momento (junto con la sintaxis del comando) simplemente con el comando `HELP`.

```
Introduce command: HELP
Available commands:
· HELP
        -> shows all commands
· SIGN_OUT
        -> closes the connection with the Termail server
· LIST_USERS
        -> sends to Termail server a request to get the users' list
· SEND_MSG <Username> <Subject> <Message>
        -> sends message to a given user
· LIST_MSGS
        -> lists all your received messages
· READ_MSG <Message ID>
        -> reads message with the given ID
Introduce command:
```

Por ejemplo, vamos a enviar un mensaje a otro usuario: utilizamos el comando `SEND_MSG [name] [message_subject] [message_text]`.

```
Introduce command: SEND_MSG Bob This_is_the_subject This a normal message sent by Alice to Bob.
 She's just testing this stunning mail service!
Message delivered successfully to 'Bob'
Introduce command:
```

Ahora Bob si inicia sesión y ejecuta el comando `LIST_MSGS` podrá ver su lista de mensajes recibidos:

```
Introduce command: LIST_MSGS
[0] - Alice: This_is_the_subject

Introduce command:
```

Finalmente, podrá leer al completo un mensaje en concreto con el comando `READ_MSG [message_id]`

```
Introduce command: READ_MSG 0
From: Alice
To: Bob
Subject: This_is_the_subject
Message: This a normal message sent by Alice to Bob. She's just testing this stunning mail service!

Introduce command:
```

Recordamos que a la vista del usuario común no se puede percibir nada fuera de lo común, pero por debajo, el software está cifrando, descifrando y verificando cada mensaje con el esquema híbrido expuesto en la anterior sección.

El código fuente puede verse y descargarse en el siguiente enlace:
https://github.com/AlejandroSantorum/Termail

5

Extensiones

Una de las partes más importantes de un proyecto es conocer sus debilidades y qué se puede hacer para mejorarlo en un futuro próximo.
Una de las posibles mejoras que podría implementarse próximamente sería que el servidor guardase toda la información de los usuarios registrados en el disco duro y no en la memoria RAM, con vista a tener un mayor servicio. Esto traería consigo un guardado de las contraseñas de los usuarios más seguro, utilizando *salt* y sus correspondientes *hashes*.

Por otro lado, otra posible mejora, sería diseñar una interfaz gráfica (web o de escritorio) para el usuario raso común. No obstante se perdería la esencia del nombre y tendríamos que buscar otra alternativa (quizá *Commandmail*?).

Finalmente, siempre estará disponible la opción de aumentar el número de comandos soportados. Solo faltaría tener la idea o la necesidad.

6

Bibliografía

- Código fuente, *Implementación de lo descrito anteriormente en Python3*. https://github.com/AlejandroSantorum/Termail
- Christof Paar & Jan Pelzl, *Understanding Cryptography*, 2010.
- Thomas Shemanske, *Modern Cryptography and Elliptic Curves*, 2017.
- Hoffstein, Pipher & Silverman, *An Introduction to Mathematical Cryptography*, 2008.
- Wikipedia, *Diffie Hellman key exchange*. https://en.wikipedia.org/wiki/Diffie-Hellman_key_exchange
- Wikipedia, *Elliptic curve cryptography*. https://en.wikipedia.org/wiki/Elliptic-curve_cryptography
- Wikipedia, *Supersingular isogeny key exchange*. https://en.wikipedia.org/wiki/Supersingular_isogeny_key_exchange
- Wikipedia, *Isogeny*. https://en.wikipedia.org/wiki/Isogeny
- PyCryptodome, *Self-contained Python package of low-level cryptographic primitives*. https://pycryptodome.readthedocs.io/en/latest/src/api.html
- Curso Redes de Comuncación II, *Material educativo del curso de Redes de Comunicación II de la Escuela Politécnica Superior de Madrid, impartido por Óscar Delgado*