Universidad Nacional Autónoma de México
Facultad de Ciencias - Ciencias de la Computación
Criptografía y seguridad 2018-1

Proyecto final

En esta práctica se realizará un chat con cifrado extremo a extremo (end to end). El protocolo usado es una simplificación del protocolo Signal, usado en aplicaciones como Signal, Whatsapp, Facebook Messenger, Google Allo, entre otros.

Tu tarea será implementar el lado del cliente, de manera que se puedan intercambiar mensajes cifrados con otros usuarios. Se usará una arquitectura de cliente-servidor para comunicar los mensajes. El servidor solo se encargará de almacenar las llaves públicas y de enviar los mensajes cifrados entre los usuarios.

Se usará Python junto con la biblioteca de algoritmos criptográficos PyNaCl. Si prefieres usar otro lenguaje, pregunta al ayudante.

1. Protocolo

1.1. Funciones criptográficas

Se usarán varias funciones, tanto de llave pública como de llave privada y funciones hash. Todas se obtendrán de la biblioteca PyNaCl, así que no es necesario implementar ningún algoritmo criptográfico. Para facilitar la implementación se describen algunas funciones y objetos que aparecen más adelante.

- Llaves. Se usarán llaves públicas y privadas, algunas para intercambio DH, otras para firmar y otras para encriptar mensajes. PyNaCl tiene métodos para crearlas fácilmente.
- $DH(priv, pub)$. Se refiere a un intercambio de Diffie-Hellman. Recibe una llave privada *priv* y una llave pública *pub*, y devuelve una cadena de bytes. En PyNaCl se usa un objeto Box y la función *shared_key*.
- Firmas digitales. Se hará uso de firmas digitales para comprobar autenticidad en la comunicación. En PyNaCl se encuentran las funciones del módulo `nacl.signing`.
- Bytes aleatorios. Siempre que se necesiten bytes aleatorios se puede usar la función `nacl.utils.random()`.

1.2. Registro

Cuando un usuario se quiere registrar, mandará al servidor un nombre de usuario junto con tres llaves públicas.

- Llave de identidad, IK.
- Llave de identidad para firmar, IKS.
- Llave de medio uso, MK, firmada con la llave anterior.

1.3. Crear sesión con otro usuario

Del lado del iniciador

Cuando Alice y Bob quieren comunicarse, primero deben crear una sesión. Supongamos que Alice es la iniciadora de la sesión. Primero solicita al servidor las llaves de Bob. El servidor devuelve IK_B (llave de identidad) y MK_B (llave de medio uso).

Alice genera una pareja de llaves efímera EK_A y calcula

$$\texttt{secreto} = \mathrm{DH}(IK_A, MK_B) \,||\, \mathrm{DH}(EK_A, IK_B) \,||\, \mathrm{DH}(EK_A, MK_B).$$

A partir de este secreto Alice puede generar llaves para cifrar los mensajes, aun si Bob todavía no ha establecido la sesión. Para que Bob pueda obtener el valor secreto, es necesario que Alice le mande las llaves EK_A e IK_A, así que Alice incluirá ambas llaves en cada mensaje enviado hasta que obtenga respuesta de Bob, que es cuando se puede confirmar que se estableció la sesión.

Del lado del receptor

Cuando Bob recibe un mensaje de Alice por primera vez, este mensaje contiene las llaves públicas EK_A e IK_A para establecer la sesión. Bob puede usar sus llaves y las de Alice para obtener

$$\texttt{secreto} = \mathrm{DH}(MK_B, IK_A) \,||\, \mathrm{DH}(IK_B, EK_A) \,||\, \mathrm{DH}(MK_B, EK_A).$$

Una vez que tiene el secreto, puede derivar las mismas llaves que Alice para cifrar los mensajes.

1.4. Cifrado de mensajes

Una vez que se tiene el valor `secreto` se pueden empezar a cifrar los mensajes. Cada mensaje será cifrado con una nueva llave. Alice (la iniciadora de la sesión) obtendrá sus llaves de la siguiente manera

$$k_{\text{enviar}}, k_{\text{recibir}} = \mathrm{SHA512}(\texttt{secreto}),$$

donde k_{enviar} son los primeros 256 bits (32 bytes) del valor hash, y k_{recibir} son los últimos 256 bits del hash. Bob hará lo mismo, pero con k_{enviar} y k_{recibir} intercambiadas.

La llave k_{enviar} será usada para cifrar los mensajes enviados, mientras que k_{recibir} será usada para descifrar los mensajes recibidos.

Cada vez que se necesite una nueva llave para cifrar, esta se obtendrá actualizando

$$k_{\text{enviar}} = \mathrm{SHA256}(k_{\text{enviar}}).$$

De forma análoga se va actualizando la llave k_{recibir}.

El cifrado se hará con el objeto SecretBox de PyNaCl.

2. Especificación de la comunicación

Toda la comunicación entre usuarios será por medio de un servidor. Cada vez que un usuario A quiera enviar un mensaje al usuario B, se enviará el mensaje al servidor y este lo entregará cuando B lo solicite (es decir, no será una comunicación en tiempo real). La comunicación será por medio de TCP, usando los sockets de Python.

Cada vez que un cliente quiera establecer comunicación con el servidor, primero deberá autenticarse. Inicialmente el cliente *saluda* al servidor, luego el servidor le mandará un mensaje *reto* de bytes

aleatorios, que el cliente tendrá que devolver, firmado con su llave de identidad. Si el cliente demuestra ser quien dice ser, el servidor aceptará la conexión, en caso contrario la cerrará.

Una vez establecida la conexión con el servidor, el cliente puede enviar y recibir los *paquetes* que se describen en las siguientes secciones.

2.1. Formato de los mensajes

Para no confundir los mensajes entre usuarios con los mensajes que se envían al servidor, usaremos la palabra *paquete* para referirnos a estos últimos. Un paquete es una secuencia de bytes que describe varios atributos, similar a un objeto de una clase.

Los mensajes están en formato JSON para ser transportados fácilmente sin preocuparse por la codificación o parseo. Un objeto de JSON se puede transformar en un diccionario de Python, y así se obtienen de inmediato los atributos del mensaje. Un paquete será un objeto JSON junto con la longitud de dicho objeto. Así se verá un paquete

`XY{'atributo': valor, ... }`

Los tres puntos se refieren a los demás atributos que puede tener el mensaje. `XY` son dos bytes que indican el tamaño del mensaje. El tamaño del mensaje es el número de caracteres contando desde { hasta el } de cierre del objeto JSON.

Por ejemplo, se quiere enviar el mensaje

`{'tipo': 'datos_usuario', 'nombre': 'Alicia'}`

Ya que tiene 43 caracteres, se le agregan los bytes `0x002b` (dos bytes) al inicio. Así, el receptor podrá leer primero dos bytes, y con esto sabrá que tiene que leer 43 bytes para tener el paquete completo.

Nota que dos bytes nos permiten mensajes de hasta 65535 caracteres.

Los atributos que son secuencias de bytes (no texto), como las llaves o los textos cifrados, serán codificados en Base64 para facilitar el transporte. Así, cada objeto JSON solo tendrá cadenas con caracteres imprimibles.

Mensajes enviados por el cliente

```
{
  'tipo': 'registro',
  'nombre': nombre,
  'llave_identidad': llave_identidad,
  'llave_identidad_firmar': llave_identidad_firmar,
  'pre_llave_firmada': pre_llave_firmada
}
```

Mensaje para solicitar el registro de un nuevo usuario en el servidor.

nombre (cadena). Es el nombre que se quiere registrar. Si el servidor ya lo tiene registrado, rechazará este registro.

llave_identidad (base64). La llave pública de identidad, se usará en el intercambio de DH con los demás usuarios. Corresponde a los bytes de un objeto PublicKey de NaCl, codificados en Base64.

llave_identidad_firmar (base64). Una llave pública de identidad, en este caso para firmar mensajes. Corresponde a los bytes de un objeto VerifyKey de NaCl, codificados en Base64.

pre_llave_firmada (base64). Una llave pública que irá firmada por la llave anterior. Corresponde a los bytes de un objeto SignedMessage de NaCl (este contiene el mensaje y la firma), codificados en Base64.

```
{
  'tipo': 'saludo',
  'nombre': nombre
}
```

Mensaje de saludo para el servidor. Sirve para iniciar la comunicación.

nombre (cadena). Nombre de usuario para ser autenticado.

```
{
  'tipo': 'telofirmo',
  'reto_firmado': reto_firmado
}
```

Mensaje para enviar el reto de autenticación, firmado con la llave de identidad.

reto_firmado (base64). El mensaje que espera el servidor ya firmado. Son los bytes de un objeto SignedMessage de NaCl, codificados en Base64.

```
{
  'tipo': 'msj',
  'nombre_destino': destino,
  'mensaje': mensaje,
  'llave_efimera': llave_efimera,
  'llave_identidad': llave_identidad
}
```

Mensaje que contiene un texto cifrado que será entregado a un usuario destino. En caso de no haber iniciado una sesión de chat con el usuario, se deberán incluir tanto la llave efímera como la llave de identidad del iniciador.

destino (cadena). Nombre de usuario a quien se dirige el mensaje.

mensaje (base64). Mensaje cifrado, codificado en Base64.

llave_efimera (base64). Opcional. Sirve para crear una nueva sesión con el usuario destino. Debe ser incluida cuando no hay dicha sesión, y una vez que se establece la sesión puede omitirse. Es una llave pública para hacer DH, bytes codificados en Base64.

llave_identidad (base64). Opcional. Llave de identidad del iniciador. El formato es como el de la llave efímera.

```
{
  'tipo': 'datos_usuario',
  'nombre': nombre_usuario
}
```

Mensaje para solicitar las llaves (públicas) de un usuario.

nombre_usuario (cadena). Nombre del usuario del que se requieren las llaves.

```
{
  'tipo': 'dame_mensajes'
}
```

Mensaje para solicitar los mensajes pendientes en el servidor.

Mensajes enviados por el servidor

El servidor responderá a cada mensaje que envíe el cliente. Cualquier respuesta del servidor contiene el atributo «ok» para indicar si la petición y la respuesta son correctas o si hubo un error en algún momento

```
{
  'ok': True,
  'detalles': texto_opcional
}
```

Mensaje para confirmar que el servidor recibió el mensaje y se pudo procesar correctamente, pero no contiene información adicional.

texto_opcional (cadena). Un texto para indicar lo que acaba de procesar el servidor, puede servir para depuración.

```
{
  'ok': False,
  'detalles': causa_de_error
}
```

Mensaje para indicar que hubo un error en lo solicitado por el cliente, por ejemplo cuando se quiere registrar pero ya existía el nombre de usuario, o cuando se quiere mandar un mensaje a un usuario que no existe.

causa_de_error (cadena). Una breve explicación de lo que posiblemente provocó el error.

```
{
  'ok': True,
  'tipo': 'reto',
  'reto': reto_a_firmar
}
```

Cuando el usuario se está autenticando, es la cadena de bytes que debe firmar el usuario para comprobar su identidad.

reto_a_firmar (base64). Cadena de 16 bytes aleatorios codificados en base64.

```
{
  'ok': True,
  'tipo': 'msjs_nuevos',
  'mensajes': lista_mensajes
}
```

Es la respuesta cuando un usuario solicita sus mensajes que están pendientes en el servidor.

lista_mensajes (lista). Es una lista (posiblemente vacía) que contiene los mensajes del usuario. Cada elemento de la lista es un diccionario de Python que contiene el remitente del mensaje, la hora de envío, el mensaje cifrado, y opcionalmente las llaves para establecer la sesión entre usuarios. **Importante:** el servidor no verifica nada sobre los mensajes cifrados ni sobre la sesión entre usuarios, todo es responsabilidad de cada cliente.

```
{
  'ok': True,
  'tipo': 'llaves',
  'pre_llave': pre_llave,
  'llave_identidad': llave_identidad
}
```

Es la respuesta cuando se solicitan las llaves públicas de un usuario. Las llaves van codificadas como en los otros mensajes, es decir, cadenas de bytes en base64.

2.2. Suposiciones buena onda

1. El servidor es honesto y ejecuta todo como se debe.
2. Los mensajes llegan en orden.
3. Solo uno de los dos (Alice o Bob) inicia la sesión, no ambos al mismo tiempo.

3. Entrega

- La práctica puede hacerse entre dos personas o de forma individual.
- Organiza tus archivos en un directorio y comprímelo en un zip. Tanto el directorio como el archivo zip tendrán un nombre de la forma `Practica4_Lopez` o `Practica4_Lopez_Juarez`, que indican el apellido paterno de una o dos personas, respectivamente.
- Sube tu archivo zip en el siguiente formulario: `https://goo.gl/forms/zTr81W6iBbo36Za32`
- Fecha límite: lunes 15 de enero.
- No se aceptará por correo.