Práctica 2 - Seguridad y criptografía

Sitio:	MOODLE DE GRADO
Curso:	REDES DE COMUNICACIONES II
Libro:	Práctica 2 - Seguridad y criptografía
Imprimido por:	Javier Martinez Rubio
Día:	miércoles, 27 de marzo de 2019, 11:11

Tabla de contenidos

1 Introducción

El objetivo de este segundo trabajo del curso es llevar a la práctica todos los conocimientos sobre seguridad y criptografía estudiados en teoría. Para ello se ha desarrollado un servicio de almacenamiento seguro de ficheros, llamado **SecureBox**, accesible mediante una API REST.

Este servicio permite recibir y enviar ficheros cifrados y firmados, para lo que se deberá diseñar e implementar un cliente (en Python 3) que consuma este servicio y permita una serie de acciones desde la línea de comandos.

En líneas generales, el servicio SecureBox aporta dos grandes funcionalidades:

- **Repositorio de identidades**, al estilo de un servidor de claves PGP. En este almacén los usuarios pueden registrar sus identidades (clave pública y datos de identificación), de forma que otros usuarios puedan buscarles, recuperar su clave pública y enviarles archivos.
- **Almacén de archivos**. Los archivos anteriores no se envían directamente al usuario destinatario, sino que son almacenados en el servidor, para que éste pueda recogerlos posteriormente.

Por tanto, en esta práctica se deberá desarrollar un cliente de línea de comandos que consuma el servicio de SecureBox y que, a grandes rasgos, permita:

- Gestionar la identidad (crear, exportar, buscar y borrar) de un usuario, realizando las llamadas adecuadas al API de SecureBox. Un usuario solo podrá disponer de una identidad en cada momento.
- Cifrar y firmar archivos de forma local. En el primer caso, se deberá especificar la identidad del destinatario.
- Enviar un archivo al servicio SecureBox, que deberá haber sido previamente cifrado y firmado.
- Recibir un archivo almacenado en SecureBox, comprobando tras su descarga su firma digital.

1.1 Desarrollo de la práctica

Python

Puesto que la práctica debe desarrollarse en Python, es conveniente tener ya cierta soltura con el lenguaje antes de empezar a codificar. Para ello, se darán sesiones en las clases de prácticas, y se subirá material al aula virtual.

Depuración

Python, a pesar de ser un lenguaje interpretado, puede (Y DEBE) ser depurado igual que C. De nuevo insistimos en que una práctica con ésta NO debe desarrollarse a base de prints y chapuzas similares. No solo porque no es correcto, sino porque es mucho más lento. Existen diversas alternativas para IDEs para Python, elige el que te resulte más cómodo y pierde el miedo a depurar.

Orden de implementación

Es importante ordenar adecuadamente el desarrollo de las distintas características que debe cumplir la práctica para no retrasarte innecesariamente:

- Como regla general, ataca el problema por partes. Por ejemplo, sería buena idea codificar las funciones que cifran y firmar ficheros aparte, hasta estar seguro de que funcionan correctamente. Luego, integra el código en el resto del programa, lo que facilita y agiliza la depuración. El mismo razonamiento aplica a otros componentes de la práctica: hazte una lista, idéntificalos y diséñalos y codíficalos por partes.
- Una buena forma de empezar a "jugar" con el API y comprender su funcionamiento es con el comando `curl`. Por ejemplo, la siguiente sentencia utiliza endpoint para obtener la clave pública de un usuario:

  ```
  # curl --verbose -H "Authorization: Bearer AaFBe2d7894C1D63" -H "Content-Type: application/json" --data '{"userID":"mi_nia"}' -X POST http://vega.ii.uam.es:8080/api/users/getPublicKey
  ```

  Podrás hacer llamadas similares, cambiando los parámetros necesarios, para entender el formato concreto de cada método del API, antes de codificar esas llamadas en el cliente. Provoca también errores para ver cómo responde el API.

2 Funcionalidad

Como ya se ha comentado, el cliente del servicio SecureBox deberá interaccionar con el servidor, con el fin de permitir subir y descargar archivos firmados y cifrados.

Para ello, el cliente deberá permitir:

- Gestionar (importar, listar y borrar) una identidad digital asociada a un único token de autenticación.
- Cifrar y firmar un archivo, dirigido a algún otro usuario del sistema.
- Enviar un archivo al servicio SecureBox.
- Recibir un archivo almacenado en SecureBox, comprobando tras su descarga su firma digital, y descifrando su contenido.

Para poder hacer estas operaciones criptográficas, es necesario crear una identidad digital previa, que nos proporcione un par de claves pública y privada.

Funcionalidad del cliente

Las opciones de la línea de comandos que debe soportar el cliente son:

Opción	Descripción
Gestión de usuarios e identidades	
--create_id *nombre email* [*alias*]	Crea una nueva identidad (par de claves púlica y privada) para un usuario con nombre *nombre* y correo *email*, y la registra en SecureBox, para que pueda ser encontrada por otros usuarios. *alias* es una cadena identificativa opcional.
--search_id *cadena*	Busca un usuario cuyo nombre o correo electrónico contenga *cadena* en el repositorio de identidades de SecureBox, y devuelve su ID.
--delete_id *id*	Borra la identidad con ID *id* registrada en el sistema. Obviamente, sólo se pueden borrar aquellas identidades creadas por el usuario que realiza la llamada.
Subida y descarga de ficheros	
--upload *fichero*	Envia un fichero a otro usuario, cuyo ID es especificado con la opción --dest_id. Por defecto, el archivo se subirá a SecureBox firmado y cifrado con las claves adecuadas para que pueda ser recuperado y verificado por el destinatario.
--source_id *id*	ID del emisor del fichero.
--dest_id *id*	ID del receptor del fichero.
--list_files	Lista todos los ficheros pertenecientes al usuario
--download *id_fichero*	Recupera un fichero con ID *id_fichero* del sistema (este ID se genera en la llamada a *upload*, y debe ser comunicado al receptor). Tras ser descargado, debe ser verificada la firma y, después, descifrado el contenido.
--delete_file *id_fichero*	Borra un fichero del sistema.
Cifrado y firma de ficheros local	
--encrypt *fichero*	Cifra un fichero, de forma que puede ser descifrado por otro usuario, cuyo ID es especificado con la opción --dest_id.
--sign *fichero*	Firma un fichero.
--enc_sign *fichero*	Cifra y firma un fichero, combinando funcionalmente las dos opciones anteriores.

Ejemplos de uso

Imaginemos dos usuarios, de nombres Pablo López, con ID GGHHII y Antonio Chicharro, de ID XXYYZZ, desean enviarse un archivo de forma segura a través de SecureBox. Suponiendo que ambas identidades, por supuesto, han sido previamente creadas y registradas en el sistema, la secuencia de comandos que deberían seguir es similar a la siguiente:

```
# python securebox_client.py --search_id Antonio
Buscando usuario 'Antonio' en el servidor...OK
3 usuarios encontrados:
[1] Antonio Chicharro, antonio.chicharro@estudiante.uam.es, ID: XXYYZZ
[2] Antonio López, antonio.lopez@estudiante.uam.es, ID: AABBCC
[3] Juan Antonio Barrera, juanantonio.barrera@estudiante.uam.es, ID: DDEEFF
```

En este momento sería necesario asegurarse de que el ID del Antonio que buscamos es XXYYZZ, confirmándolo con él por otro medio. Una vez satisfechos, podríamos ya enviarle un fichero a través de SecureBox:

```
 # python securebox_client.py --upload foto1.jpg --dest_id XXYYZZ
Solicitado envio de fichero a SecureBox
-> Firmando fichero...OK
-> Recuperando clave pública de ID XXYYZZ...OK
-> Cifrando fichero...OK
-> Subiendo fichero a servidor...OK
Subida realizada correctamente, ID del fichero: AABBCCDDEEFF
```

A partir de este momento, el fichero está disponible en el servidor para que sea recuperado por Antonio. Para ello, tendríamos que comunicarle el ID del fichero, AABBCCDDEEFF, para que el pudiera recuperarlo con un comando similar al siguiente:

```
# python securebox_client.py --download AABBCCDDEEFF --source_id GGHHII
Descargando fichero de SecureBox...OK
-> 33245 bytes descargados correctamente
-> Descifrando fichero...OK
-> Recuperando clave pública de ID GGHHII...OK
-> Verificando firma...OK
Fichero descargado y verificado correctamente
```

2.1 Gestión de identidades y usuarios

Esta funcionalidad hace referencia a la capacidad del cliente para gestionar la identidad digital del usuario, de forma que quede registradas en SecureBox. Esta identidad deberá ser almacenada localmente por el cliente, típicamente en forma de fichero o ficheros en un directorio.

Concretamente, el cliente deberá poder:

- **Crear una nueva identidad**, con la opción `--create_id [alias]`. El cliente deberá generar un nuevo par de claves pública y privada, con el alias adecuado (si se ha especificado alguno), y registrarla en SecureBox con la llamada adecuada, lo que generará un ID único para el usuario. **En cada momento, cada usuario sólo puede tener una identidad activa asociada a un token de autenticación**. Si se crea y registra una nueva identidad, los datos asociadas a la antigua (nombre, email y clave pública) se perderán.

```
# python securebox_client.py --create_id "Mi primera identidad"
Generando par de claves RSA de 2048 bits...OK
Identidad con ID#01020304 creada correctamente
```

- **Buscar una identidad** existente, con la opción `--search_id`. Esto permite encontrar un usuario del que sólo sabemos su nombre o correo, por ejemplo, y buscar su ID para poder enviarle un archivo de forma segura.

```
# python securebox_client.py --search_id "Antonio"
Buscando usuario 'Antonio' en el servidor...OK
3 usuarios encontrados:
[1] Antonio Chicharro, antonio.chicharro@estudiante.uam.es, ID: XXYYZZ
[2] Antonio López, antonio.lopez@estudiante.uam.es, ID: AABBCC
[3] Juan Antonio Barrera, juanantonio.barrera@estudiante.uam.es, ID: DDEEFF
```

- **Borrar una identidad** existente, con la opción `--delete_id`. Obviamente, sólo es posible borrar identidades creadas por el propio usuario.

```
# python securebox_client.py --delete_id 01020304
Solicitando borrado de la identidad #01020304...OK
Identidad con ID#01020304 borrada correctamente
```

2.2 Cifrado y firma de archivos

Como se trata de un servicio seguro, los ficheros deberán ser subidos al servidor firmados y cifrados, utilizando la petición del API adecuada. Para ello, deberá utilizar un esquema híbrido, con el orden correcto para la firma y cifrado, el uso de clave de sesión, etc...

Para que todos los archivos protegidos sean compatibles entre distintos usuarios, el cliente deberá utilizar las siguientes primitivas criptográficas:

- *Cifrado simétrico*: AES con modo de encadenamiento CBC, con IV de 16 bytes, y longitud de clave de 256 bits.
- *Función hash*: SHA256
- *Cifrado asimétrico*: RSA con longitud de clave de 2048 bits.

Por otro lado, los adjuntos al mensaje (tanto firma como sobre digital) se concatenarán delante del mismo.

El cliente deberá también poder cifrar y firmar archivos de forma autónoma, sin que éstos sean subidos a SecureBox, con las opciones `--encrypt`, `--sign` y `--enc_sign`, para cifrar, firma y cifrar y firmar, respectivamente.

2.3 Envío y descarga de ficheros

El envío de ficheros constituye la principal funcionalidad de SecureBox, y su objetivo es poder enviar ficheros de forma segura a otro usuario. Por ejemplo, para hacerlo con el usuario con ID 34FAB7 podría utilizarse un línea de comandos como la siguiente:

```
# secure_client --dest_id 34FAB7 --upload doc_secreto.pdf
Solicitado envio de fichero a SecureBox
-> Firmando fichero...OK
-> Recuperando clave pública de ID 34FAB7...OK
-> Cifrando fichero...OK
-> Subiendo fichero a servidor...OK
Subida realizada correctamente, ID del fichero: 03AAB3
```

En este momento, el usuario receptor podría recuperar el fichero con el siguiente comando:

```
# secure_client --download 03AAB3 --source_id GGHHII
Descargando fichero de SecureBox...OK
-> 33245 bytes descargados correctamente
-> Descifrando fichero...OK
-> Recuperando clave pública de ID GGHHII...OK
-> Verificando firma...OK
Fichero 'doc_secreto.pdf' descargado y verificado correctamente
```

2.4 Descripción del API

En esta sección se describe con detalle las distintas funciones que proporciona el API del servidor, que deberán ser llamadas adecuadamente por el cliente para implementar la funcionalidad requerida.

Como ya se ha comentado, el API se ha diseño en forma de API REST. Asegúrate de entender bien en qué consiste, y sus principales ventajas antes de continuar [1][2].

Autenticación

En las aplicaciones Web tradicionales, la autenticación de los usuarios se realiza normalmente con esquemas de usuario/contraseña, y almacenando luego el ID del usuario en una sesión (que suele implementarse con cookies en el lado del cliente). Cuando el usuario visita de nuevo una página que necesita autenticación, el navegador envia la cookie al servidor, éste busca la ID del usuario en la BD y, si su sesión no ha expirado, permite el acceso sin tener que volver a introducir las credenciales.

Con un API, puesto que suelen estar diseñadas para ser llamadas desde aplicaciones, y no desde un navegador, el uso de sesiones y esquemas de usuario/contraseña no es la mejor opción. La solución más común es utilizar autenticación basada en tokens.

El funcionamiento es muy sencillo: el usuario se autentica una única vez con su usuario/contraseña, o un mecanismo similar, y la aplicación devuelve un token único para ese usuario que deberá ser enviada en cada petición al API posterior. Un estándar muy común que implementa este método, utilizado hoy en día por prácticamente todas las APIs, se llama OAuth, específicamente su versión 2.

Para empezar a utilizar las funciones del API de la práctica, deberás, por tanto, solicitar un token de autenticación primero. Para ello dirígete a la URL:

```
http://vega.ii.uam.es:8080
```

Allí se solicitarán unos datos, que deberás rellenar. Recibirás entonces en el correo especificado el token, que deberás incluir en una cabecera `HTTP Authorization` de todas las llamadas al API. Concretamente, si el token que has recibido es AaFBe2d7894C1D63, la cabecera HTTP a incluir quedaría:

```
Authorization: Bearer AaFBe2d7894C1D63
```

Endpoints

Las diferentes funciones o métodos que proporciona un API para que sean llamados reciben el nombre de *endpoints*. En el API de SecureBox, existen los siguientes:

Funciones relacionadas con la gestión de identidades

- */users/register* - registra un usuario en el sistema
- */users/getPublicKey* - obtiene la clave pública de un usuario
- */users/search* - obtiene datos de un usuario por nombre o correo electrónico
- */user/delete* - borrar un usuario

Funciones relacionadas con la gestión de ficheros

- */files/upload* - sube un fichero al sistema
- */files/download* - descarga un fichero
- */files/list* - lista todos los ficheros pertenecientes a un usuario
- */files/delete* - borra un fichero

Todas estas funciones cuelgan de la URL *https://vega.ii.uam.es:8080/api*, de forma que la llamada para subir un fichero, por ejemplo, quedaría de la siguiente forma:

```
https://vega.ii.uam.es:8080/api/files/upload
```

Ten en cuenta, también, que todos los endpoints del API esperan recibir los datos en formato JSON, así que asegúrate de enviarlos correctamente. Por ejemplo, si utilizas la librería *requests* de Python, puedes utilizar un código similar al siguiente:

```
url = 'https://vega.ii.uam.es:8080/api/users/getPublicKey'
args = {'idUser': 123}
r = requests.post(url, json=args)
print(r.text)
```

Gestión de identidades

Estas funciones sirven para la gestión de las identidades de los usuarios que utilicen *SecureBox*. A grandes rasgos, por tanto, para empezar a poder utilizar una identidad el cliente deberá:

- Obtener un token de autenticación
- Crear un par de claves público/privada
- Registrar el usuario en el sistema, con su nombre, correo y clave pública

Registro de usuarios - /users/register

Argumentos:

- **nombre**: nombre completo del usuario, para que pueda ser buscado después por otros usuarios.
- **email**: correo electrónico.
- **publicKey**: clave pública del usuario, que será utilizada por otros usuarios para enviarle ficheros cifrados. Deberá utilizarse el formato PEM.

Respuesta

Como todos los endpoints, la respuesta del API será en formato JSON, con los siguientes parámetros:

- **nombre**: nombre completo del usuario.
- **ts**: marca de tiempo (timestamp), en formato UNIX, de registro del usuario.

Posibles errores: Ninguno

Ejemplo

Petición

```
POST https://vega.ii.uam.es:8080/api/users/register HTTP/1.1
Content-Type: application/json
Content-Length: 102
Authorization: Bearer AaFBe2d7894C1D63

{'nombre': 'Antonio Chicharro', 'email': 'antonio.chicharro@estudiante.uam.es', 'publicKe
y': '---BEGIN ....'}
```

Respuesta

Si la petición es correcta, el servidor devuelve un código HTTP `200 OK` junto con información adicional:

```
HTTP/1.1 200
Content-Type: application/json
{'nombre': 'Antonio Chicharro', 'ts': '1513781531.16802'}
```

Obtener clave pública - /users/getPublicKey

Obtiene la clave pública de un usuario. Típicamente esta función se llamará cuando sea necesaria encontrar esta clave para poder enviar a un usuario concreto un fichero cifrado.

Argumentos:

- **userID**: identificador único del usuario cuya clave pública solicitamos.

Respuesta

- **publicKey**: clave pública solicitada

Posibles errores: USER_ID1

Búsqueda de usuarios - /users/search

Busca los metadatos de un usuario a partir de su nombre o correo electrónico.

Argumentos: estructura JSON con los siguientes parámetros:

- **data_search**: cadena de búsqueda, que será contrastada contra el nombre y correo electrónico.

Respuesta: si no se producen errores, la respuseta del API es una estructura JSON con los siguientes parámetros:

Como todos los endpoints, la respuesta del API será en formato JSON, con los siguientes parámetros: **userID, nombre, email, publicKey** y **ts**

Posibles errores: USER_ID2

Borrado de usuarios - /users/delete

Elimina un usuario a partir de su identificador.

Argumentos: estructura JSON con los siguientes parámetros:

- **userID:** ID del usuario a ser borrado

Respuesta: si no se producen errores, la respuseta del API es una estructura JSON con los siguientes parámetros:

- **userID**: ID del usuario borrado

Posibles errores: Ninguno

Gestión de ficheros

Estas funciones gestionan el almacenamiento de ficheros en *SecureBox*. Esencialmente, el flujo de llamadas sería el siguiente:

- Un usuario A desea enviar un fichero a otro usuario B. Para ello, debe comenzar por obtener la clave pública de B.
- Preparar (cifrar y firmar) el fichero para su envío a B a través de SecureBox.
- Subir el fichero a SecureBox, lo que devuelve un identificador, *file_id*.
- Enviar el identificador a B por cualquier medio (correo electrónico, mensajería, etc.), para que éste pueda solicitar su descarga.
- B comprueba la firma del fichero y, si es correcta, descifra el contenido.

Subida de ficheros - /files/upload

Función para subir un fichero al sistema que, como se ha comentado, debe ser previamente cifrado y firmado. Otro usuario podrá, después, descargarlo, descifrarlo y verificar su firma.

Argumentos: esta es la única función de todo el API que no recibe los argumentos JSON, sino como un formulario HTML tipo POST, con nombre *ufile*.

Respuesta: la respuesta, sin embargo, sí se realizará en JSON, con los siguientes parámetros:

- **file_id**: identificador del fichero asignado por el sistema, necesario para solicitar su descarga posterior.
- **file_size**: tamaño del fichero, en bytes. El tamaño aceptado está limitado a 50KB.

Posibles errores: FILE1

Ejemplo

Con un comando como el siguiente:

```
# curl --verbose -H "Authorization: Bearer AaFBe2d7894C1D63" -F "ufile=@/home/pepito/analysis.py" https://vega.ii.uam.es:8080/api/files/upload
```

Si la respuesta es corercta, el servidor responderá con un código de error HTTP `200 OK` , como habitualmente, e información de metadatos:

```
HTTP/1.1 200
Content-Type: application/json

{"file_id": "xxxxxxxxxxxxxxxxxxxxxxxxxxxx", "file_size": "12345"}
```

Descarga de ficheros - /files/download

Función para descargar un fichero del sistema.

Argumentos: estructura JSON con los siguientes parámetros:

- **file_id**: identificador del fichero a descargar

Respuesta: si no se producen errores, la respuesta del API es directamente el contenido binario del fichero.

Posibles errores: FILE2 - El fichero no existe

Listado de ficheros - /files/list

Función para listar todos los ficheros pertenecientes a un usuario.

Argumentos: no necesita

Respuesta: si no se producen errores, la respuseta del API es una estructura JSON con los siguientes parámetros:

- **files_list**: lista con los ID de todos los ficheros pertenecientes al usuario.
- **num_files**: número total de elementos de la lista anterior.

Posibles errores: Ninguno

Borrado de ficheros - /files/delete

Función para borrar un fichero del sistema.

Argumentos: estructura JSON con los siguientes parámetros:

- **file_id**: identificador del fichero a borrar

Respuesta: si no se producen errores, la respuseta del API es una estructura JSON con los siguientes parámetros:

- **file_id**: identificador del fichero borrado

Posibles errores: FILE2 - El fichero no existe

Gestión de errores

Todos los *endpoints* del API devuelven los errores en dos niveles diferentes:

- Códigos de error HTTP en la cabecera de la respuesta HTTP
- Un objeto JSON en el cuerpo de la respuesta, con información adicional, como el siguiente ejemplo.

```
{"http_error_code": 401, "error_code": "TOK1", "description": "Token de usuario incorrecto"}
```

Los campos son bastante auto-descriptivos, pero su significado es el siguiente:

- **http_error_code**: código de error HTTP, para un primer filtrado.
- **error_code**: código de error JSON específico del API (tabla con todos los códigos a continuación)
- **description**: descripción del error en lenguaje natural. Puede (y debe) ampliarse para dar al usuario más información sobre las posibles causas del error y qué puede hacer para solucionarlo.

Por último, en la siguiente tabla se recogen todos los posibles códigos de error devueltos por el API y su significado:

Código HTTP	Código JSON	Descripción
401	TOK1	Token de usuario incorrecto.
403	TOK2	Token de usuario caducado, se debe solicitar uno nuevo.
401	TOK3	Falta cabecera de autenticación. No se ha incluido la cabecera o ésta tiene un forma incorrecto.
403	FILE1	Se ha supera el tamaño máximo permitido en la subida de un fichero (50Kb)
401	FILE2	El ID de fichero proporcinado no es correcto (el fichero no existe o el usuario no disponde permisos para acceder a él)
401	FILE3	La cuota máxima de almacenamiento de ficheros ha sido superada (20)
401	USER_ID1	El ID de usuario proporcionado no existe
401	USER_ID2	No se ha encontrado el usuario con los datos proporcionados para la búsqueda con la función `search`.
401	ARGS1	Los argumentos de la petición HTTP son erróneos

2.5 Resumen de recursos

Como es habitual en estas prácticas, puede utilizarse cualquier librería para la realización de las mismas, pero se recomiendan, al menos, las siguientes:

- Librería requests, para la realización de peticiones HTTP al API.
- Librería PyCryptodome, para todo tipo de operaciones criptográficas