2019

Gestor de contraseñas

SEGURIDAD EN EL DISEÑO DEL SOFTWARE

JOSÉ JOAQUÍN ALARCÓN CALERO
KIRIL VENTSISLAVOV GAYDAROV

Índice

1. Introducción

El objetivo de esta práctica consiste en la creación de un gestor de contraseñas (al estilo de PasswordSafe, Password Gorilla, 1Password, KeePass, etc.) con almacenaje en servidor que permita su acceso desde distintos clientes remotos.

Obligatoriamente hemos tenido que implementar:

- Arquitectura cliente/servidor.
- Cada entrada incluirá, como mínimo, un identificador, un usuario y una contraseña.
- Mecanismo de autentificación seguro (gestión de contraseñas e identidades).
- Transporte de red seguro entre cliente y servidor (se puede emplear algún protocolo existente como TLS o HTTPS).
- Cifrado de la base de datos de contraseñas en el servidor.

Y los puntos opcionales que hemos elegido implementar son:

- Generación de contraseñas aleatorias y por perfiles (longitud, grupos de caracteres, pronunciabilidad, etc.)
- Incorporación de datos adicionales (notas, números de tarjeta de crédito, etc.) en cada entrada.
- Optimización de la privacidad (conocimiento cero: el servidor sólo recibe datos cifrados por el cliente).
- Compartición de contraseñas con grupos de usuarios usando clave pública.

2. Descripción

Pasamos a detallar como hemos implementado cada punto:

- **Arquitectura cliente/servidor.**

El proyecto se divide en dos partes principales. Por un lado, está el programa que actúa de cliente. Este programa es que muestra la interfaz con la que interactúa el usuario final del sistema. En el contexto actual, la interfaz muestra un menú principal que permite la navegación hasta diferentes submenús del menú principal que a su vez permiten navegar o acceder a opciones relacionadas con la modificación, creación, lectura y borrado de diferentes elementos como tarjetas, contraseñas y notas de los diferentes usuarios.

Además de todo esto, se emplean diferentes formularios que facilitan reutilizar el código fuente ya que se llaman varias veces. Estos formularios permiten a los usuarios introducir los datos relativos a sus contraseñas, sus tarjetas, sus notas o su propio usuario.

La parte del servidor se divide en dos capas principales:

La clase que realiza el papel de CAD (Capa de acceso a datos). Esta clase contiene todos los métodos necesarios para la interacción con la base de datos que emplea el programa para su funcionamiento. En estos métodos se tratan los posibles errores que se podrían cometer a la hora de ejecutar alguna instrucción sobre la base de datos, así como la trazabilidad de las acciones que realizan los usuarios en forma de log. Esta última característica añade todos los movimientos realizados en una tabla diferente de la base de datos.

La clase principal del servidor. Contiene las variables globales que indican los parámetros de conexión a la base de datos. Su principal función es permitir la conexión simultánea de los usuarios con el sistema, así como la realización del papel de la fachada porque define varios endpoints a los que apuntarán todos los usuarios a la hora de realizar cualquier acción.

En ambas partes del sistema se han definido archivos adicionales que facilitan la reutilización del código y su desacoplamiento. Estos archivos contienen todas las funciones que se consideran secundarias y que se estima que se utilizarán a menudo.

- **Cada entrada incluirá, como mínimo, un identificador, un usuario y una contraseña.**

Hemos creado en la BBDD una tabla de usuarios en la que cuando nos registramos guardamos, el usuario, la contraseña (la dividimos y guardamos el Hash y Salt), y datos propios del usuario como son nombre, apellido y el correo. A la hora de loguearnos solo pedimos la contraseña y el usuario, y desde la contraseña podemos sacar si es la correcta o no mediante el Hash.

- **Mecanismo de autentificación seguro (gestión de contraseñas e identidades).**

Lo hacemos mediante el Hash como hemos mencionado anteriormente, en el login pedimos la contraseña y el usuario y desde ahí comprobamos que el Hash generado sea el mismo que el de la BBDD.

- **Transporte de red seguro entre cliente y servidor (se puede emplear algún protocolo existente como TLS o HTTPS).**

Para el transporte usamos en plan API con consultas de tipo POST, también generamos un certificado para comprobar que la conexión sea segura.

- **Cifrado de la base de datos de contraseñas en el servidor.**

Es importante distinguir entre la contraseña que utiliza el usuario para iniciar sesión en el sistema y las contraseñas que se crea el usuario una vez dentro del sistema para guardarlas y almacenarlas ahí.

En ambos casos, todas las contraseñas se cifran y se almacenan en la base de datos de forma que sólo el usuario, propietario de estas contraseñas, pueda leerlas y utilizarlas. Esto aporta un nivel de seguridad muy elevado ya que se incorpora el patrón de conocimiento cero dónde el servidor no conoce los datos y sólo el cliente es capaz de recuperarlos y emplearlos.

Para la tabla que almacena los usuarios del sistema, sólo se guarda en texto plano el nombre del usuario, su clave pública, el hash y la salt. Todos los campos como nombre, apellidos, email y clave privada se almacenan en una estructura encriptada cuyo contenido no puede ser leído por nadie excepto el usuario titular.

- **Generación de contraseñas aleatorias y por perfiles (longitud, grupos de caracteres, pronunciabilidad, etc.)**

Para hacer esta parte hemos utilizado la librería: *github.com/sethvargo/go-password/Password.*
La cual nos ha permitido crear contraseñas aleatorias por: longitud de contraseña, número de dígitos, número de símbolos, distinguir entre mayúsculas y minúsculas, y repetir caracteres o no.

- **Incorporación de datos adicionales (notas, números de tarjeta de crédito, etc.) en cada entrada.**

Para la incorporación de datos adicionales simplemente hemos añadido notas y tarjetas de créditos, y hemos seguido el mismo proceso que para las contraseñas.
En las notas hemos añadido una fecha y un texto.
Y en las tarjetas de crédito, el número de la tarjeta, nombre del propietario, fecha de caducidad y CCV.

- **Optimización de la privacidad (conocimiento cero: el servidor sólo recibe datos cifrados por el cliente).**

Esta es la parte que más nos ha costado hacer, ya que trastocaba cualquier cosa ya hecha.
Hemos tenido que encriptar pasar todo el código al cliente, y en la BBDD encriptar todos los campos por usuario y que cuando el cliente quiere hacer algo con estos datos, se los pide al servidor y este simplemente busca en la tabla correspondiente por el usuario y le pasa el campo data (donde se encuentran todos los datos encriptados), el cliente lo desencripta y ya lo puede usar. Una vez añadido, modificado, borrado, etc., el cliente se lo vuelve a pasar encriptado al servidor y este lo guarda en la tabla correspondiente.
Un ejemplo para que se entienda mejor, para la parte de notas:
En la BBDD tenemos una tabla llamada notes (id, user, data), el campo data este encriptado y contiene listas de struct texto y fecha. Cuando el cliente quiere añadir, modificar, eliminar o listar las notas, este le pide al servidor que nos devuelva las notas y el servidor busca el usuario que esta logueado y devuelve este campo data, y en el cliente ya se hace lo correspondiente.

- **Compartición de contraseñas con grupos de usuarios usando clave pública.**

De todas las tareas descritas anteriormente, esta fue la más difícil de conseguir por los numerosos problemas por falta de conocimientos que se han ocasionado.

El funcionamiento esta característica es el siguiente:

1- Un usuario crea una contraseña nueva para un sitio web.
2- Al crear la contraseña nueva, el sistema automáticamente generará una clave AES para esta contraseña y esta no será cambiada nunca al igual que su identificador.
3- El usuario decide compartir esa contraseña con otro usuario del sistema.
4- En la pantalla del usuario se visualizarán todos los usuarios existentes en el sistema, menos él suyo. En este caso se realizará, previamente, una llamada al servidor que, a su vez, consultará la base de datos y devolverá una lista formada por las parejas de [nombre_usuario]##[clave_publica_usuario]. Entre cada pareja de los datos anteriores se utilizarán 3 hashtags para separar los datos. Un ejemplo con dos usuarios seleccionables para compartir con ellos: pepe##clavePubPepe###juan##clavePubJuan
5- El usuario seleccionará uno de los usuarios de la lista y el programa cliente se encargará de realizar las operaciones pertinentes.
6- El primer paso es encriptar la estructura que contiene todos los datos de la contraseña usando su clave AES.
7- El segundo paso es encriptar la clave AES empleando para ello la clave pública del usuario que se seleccionado como objetivo de la compartición.
8- El tercer paso es enviar los datos al servidor para que realice la escritura en la base de datos.
9- Una vez compartida una contraseña, el usuario con el que se comparte podrá visualizarla. Para conseguir esto se han realizado las siguientes operaciones.
10- Cuando el usuario procede a visualizar todas las contraseñas, tanto suyas como compartidas con él, las primeras contraseñas que aparecen son las suyas propias durante 5 segundos.
11- Una vez terminados estos 5 segundos, el programa cliente realizará una llamada al programa servidor que a su vez consultará la base de datos para comprobar si existen contraseñas compartidas con este usuario.
12- El servidor devolverá los datos de la consulta de la base de datos al programa cliente y este se encargará de desencriptarlos para mostrar las contraseñas.
13- El primer paso que realiza es el que desencripta la clave AES, previamente encriptada con su clave pública, utilizando para tal fin su clave privada.
14- Una vez obtenida la clave AES que se utilizó para cifrar el conjunto, el programa procederá a descifrar la estructura entera que contiene todos los datos de la contraseña compartida utilizando la clave AES.
15- Por último, se empleará un vista predefinida para mostrar todas las contraseñas con un formato adecuado para facilitar su lectura por pantalla.

La compartición de las contraseñas sólo fue el comienzo. Después de compartir una contraseña, ¿qué ocurre si esta contraseña sufre alguna modificación? ¿Habrá que volver a compartirla con cada uno de los usuarios originales? La respuesta es, no.

El motivo por el cual se define y almacena la clave AES dentro de la estructura de la contraseña es para que pueda ser reutilizado en el caso de que la contraseña sufra alguna modificación.

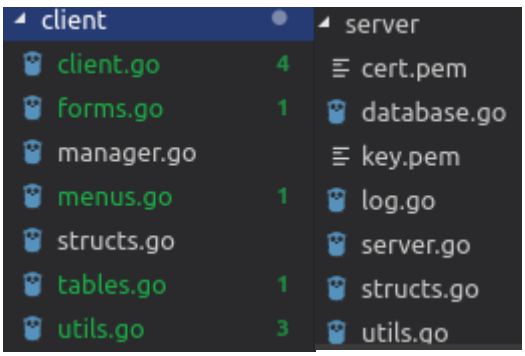
De esta forma, sólo sería necesario volver a cifrar la estructura de la contraseña y enviarla al servidor para que la almacena en la base de datos.

Esto facilita la tarea de modificar una contraseña compartida ya que los usuarios que tienen acceso a ella seguirán siendo capaces de descifrar la estructura porque la contraseña AES no ha cambiado.

3. Resultados

- **ASPECTOS DE IMPLEMENTACIÓN**

Hemos dividido el código en dos carpetas, una para cliente y otra para servidor (client and server).



Se puede observar que en las dos tenemos un archivo para las estructuras (structs.go).

También se puede ver que se repiten archivos utils.go, estos se usan para guardar funciones que se llaman varias veces desde el archivo principal. La siguiente imagen es un ejemplo de esto, en el que se ven las funciones borrar pantalla (clearScreen()) y encriptar (encrypt).

```go
//A screen cleaner. Only works on Unix based systems
func clearScreen() {
    c := exec.Command("clear")
    c.Stdout = os.Stdout
    c.Run()
}

//Encrypts the given data using AES
func encrypt(data, key []byte) (out []byte) {
    out = make([]byte, len(data)+16)    //Reserve space at the begginig of the array
    rand.Read(out[:16])                 //Reads the output array
    blk, err := aes.NewCipher(key)      //Creates a new AES cipher using the key
    chk(err)                            //Check if there's any error
    ctr := cipher.NewCTR(blk, out[:16]) //Cipher using CTR
    ctr.XORKeyStream(out[16:], data)    //Encrypt the data

    return
}
```

En la parte del servidor también podemos ver dos archivos (cert.pem y key.pem) estos son los certificados que se crean y que hay que ir actualizando cada X tiempo. También podemos observar un archivo log.go, donde llamamos a la base de datos para que inserte el método que acaba de utilizar un usuario y si se ha podido realizar o el error que ha dado en caso contrario. Esto se puede observar en la siguiente imagen.

```go
/*
    WRITES THE INPUT OF THE REQUEST AND THE OUTPUT OF THE RESPONSE FROM A FUN
*/
func writeLog(method string, username string, id int, msg string) {
    db, err := sql.Open("mysql", DB_Username+":"+DB_Password+"@"+DB_Protocol+"("+DB_IP+":"+DB_Port+")/"+DB_Name)
    if err != nil {
        consoleTimeStamp()
        fmt.Println(err.Error())
    }

    defer db.Close()

    var query = "INSERT INTO log(date, method, user, correlation, text) " +
        "VALUES(NOW(), '" + method + "', '" + username + "', '" + strconv.Itoa(id) + "', \"" + strings.Replace(msg, "'", "", -1) + "\");"

    insert, err := db.Query(query)

    if err == nil {
        defer insert.Close()
    }
}
```

En la parte del servidor también se puede ver el archivo database.go, en el que solo hace llamadas a la base de datos, como hemos dicho anteriormente al utilizar conocimiento cero, la mayoría de funciones que hay

aquí son referentes a UPDATE ya que el servidor no sabe los datos que hay, simplemente los coge y actualiza.

En la parte del cliente, se puede observar que hay un archivo llamado tables.go, que contiene las funciones para listar, los datos del usuario, tarjetas de crédito, notas y contraseñas. Esto lo hemos hecho así por optimización de código ya que es mucho más entendible.
Tenemos otro archivo llamado forms.go, en el que están todas las funciones que piden datos al usuario, como puede ser el registro como se observa en la siguiente imagen.

```
func register(username *string, password *string, name *string, surname *string, email *string) {
    fmt.Println(Bold(Green("\nRegister")))
    fmt.Println("----------------------")

    fmt.Print("Enter username: ")
    fmt.Scanln(username)

    fmt.Print("Enter password: ")
    fmt.Scanln(password)

    fmt.Print("Enter name: ")
    fmt.Scanln(name)

    fmt.Print("Enter surname: ")
    fmt.Scanln(surname)

    fmt.Print("Enter email: ")
    fmt.Scanln(email)
}
```

Por último en la parte del cliente, podemos ver un fichero llamado manager.go, en el que se encuentra lo que es el menú del usuario. Todo esto lo gestionamos mediante switch.

Los ficheros principales serían client.go y server.go desde los cuales se llaman a todos los demás.

Como se puede observar en algunas imágenes también hemos comentado todo el código por si algo no se entiende bien, esto lo hemos hecho así ya que esto es un lenguaje de programación nuevo para nosotros y al principio nos costó bastante el aprendizaje, también porque hemos tenido muy poco tiempo para realizarlo y no hemos sido muy continuados con el desarrollo.

- **OTROS ASPECTOS A DESTACAR**

Otros aspectos a destacar de la implementación son el requisito opcional de generación de contraseñas aleatorias, que como podemos observar en la siguiente imagen se nos ha quedado una función bastante larga, y en la que se piden bastantes requisitos (esto para cumplir con este requisito opcional).

```go
func addPassword() passwordsData {
    var pd passwordsData
    var random string

    pd.Modified = time.Now().Format("2006-01-02 15:04:05")
    pd.AES = generateAESkey()

    fmt.Print("Enter the URL of the password's site: ")
    fmt.Scanln(&pd.Site)

    fmt.Print("Enter your username: ")
    fmt.Scanln(&pd.Username)

    fmt.Print("Would you like to generate a random password for it? (y/n): ")
    fmt.Scanln(&random)

    if random == "y" {
        var size, nDigits, nSymbols int
        var choice string
        var upperLower = false
        var repeat = false

        fmt.Print("Size of the password: ")
        fmt.Scanln(&size)

        fmt.Print("Number of digits: ")
        fmt.Scanln(&nDigits)

        fmt.Print("Number of symbols: ")
        fmt.Scanln(&nSymbols)

        fmt.Print("Allow upper and lowercase letters? (y/n): ")
        fmt.Scanln(&choice)

        if choice == "y" {
            upperLower = true
        }

        fmt.Print("Repeat characters? (y/n): ")
        fmt.Scanln(&choice)

        if choice == "y" {
            repeat = true
        }

        // Generate a password that is 64 characters long with 10 digits, 10 symbols,
        // allowing upper and lower case letters, disallowing repeat characters.
        pass, err := password.Generate(size, nDigits, nSymbols, !upperLower, repeat)
        if err != nil {
            log.Fatal(err)
        }

        pd.Password = pass

        fmt.Print(Bold(Red("Showing the generated password for")), Underline(Bold(White("5 seconds!"))),
        fmt.Println(pd.Password)

        time.Sleep(5 * time.Second)
    } else {
        fmt.Print("Enter your password: ")
        fmt.Scanln(&pd.Password)
    }

    return pd
}
```

Por último, otro aspecto a destacar de la implementación es que para el control de la sesión no hemos utilizado ningún token, hemos implementado de tal forma que cuando un usuario inicia sesión, este inicio de sesión se le envían a las demás funciones que hay dentro de la parte privada.

```
func managePasswords(client *http.Client, username string) {
```

Como se puede observar en la anterior imagen vamos pasando el cliente que se ha logueado de una función a otra.

- **INTERFAZ**

Para la interfaz no hemos elegido ninguna interfaz gráfica en sí, hemos usado la terminal con alguna librería para que quede más visible y funcional, ya que consideramos que la importancia de esta práctica no es el aspecto visual sino la seguridad que se le ofrece al usuario.

```
Awaiting connections through default port: 8080
[2019-05-29 16:57:26]: 192.168.100.16:34240 opened a new connection
[2019-05-29 16:58:07]: 192.168.100.16:33861 opened a new connection
^[[2019-05-29 17:50:35]: 192.168.100.16:46996 closed the existing connection
```

En la figura anterior se puede observar las peticiones que le llegan al servidor.

```
--------------------------
|       MasterPASS       |
--------------------------

[ 1 ] Login
[ 2 ] Register
[ 3 ] Exit
Choice:
```

En la figura anterior se puede observar la pantalla de inicio de nuestra aplicación, donde puedes loguearte, registrarte o salir de ella. Esto sería la parte pública de la aplicación, ya que todo lo demás es privado.

```
--------------------------
|       Main menu        |
--------------------------

Welcome a.

[ 1 ] Manage passwords
[ 2 ] Manage cards
[ 3 ] Manage notes
[ 4 ] User settings
[ 5 ] Logout
Choice:
```

En la figura anterior se puede observar la pantalla de inicio de nuestra aplicación (después de haberse logueado, parte privada), donde se puede ver que puedes gestionar tu opciones de usuario, las contraseñas, tarjetas de crédito y notas.

- **BASE DE DATOS**

Para la base de datos hemos utilizado mysql mediante el paquete “database/sql” la cual proporciona una interfaz ligera orientada a filas. Para la implementación de la base de datos hemos utilizado la herramienta visual MySQL Workbench. Por último, cabe decir que solo empleamos seis tablas (users, cards, notes, password, log y shares).
La tabla log se utiliza para guardar un registro de lo que se hace.
La tabla share se utiliza para la compartición de contraseñas.
Las demás tablas son muy similares, ya que tienen un campo Data el cual está encriptado y contiene todos los datos de esa tabla.

Nota: Como no es una base de datos local, en el último punto de la memoria se hace referencia la usuario, contraseña y cadena de conexión que hace falta para verla.

- **ANÁLISIS DE LOS RESULTADOS**

Los resultados obtenidos después de realizar este proyecto son que es un sistema fiable con distintos mecanismos de seguridad en las diferentes acciones que permite realizar el usuario.

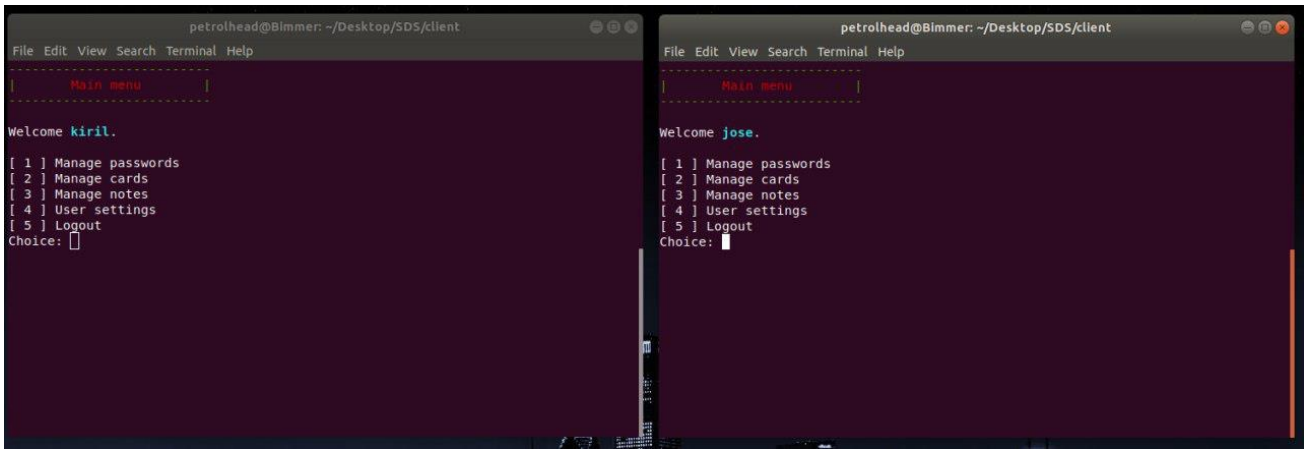
Para la arquitectura cliente/servidor la comunicación es segura, toda la información relevante transferida del cliente al servidor ha sido previamente encriptada y codificada.

Para el sistema de autenticación seguro, la contraseña del usuario pasa previamente por una función hash y luego en el servidor se le aplica bcrypt.

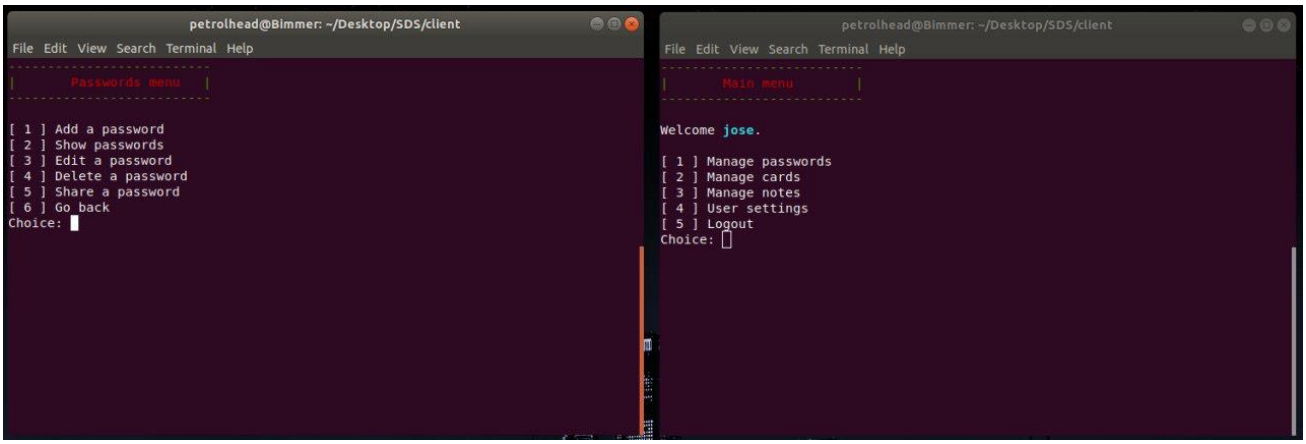
Para el cifrado con conocimiento cero, el servidor solo conoce la información necesaria para su funcionamiento. Durante el registro a la contraseña

- **EJEMPLO DE COMPARTICIÓN DE CONTRASEÑAS**

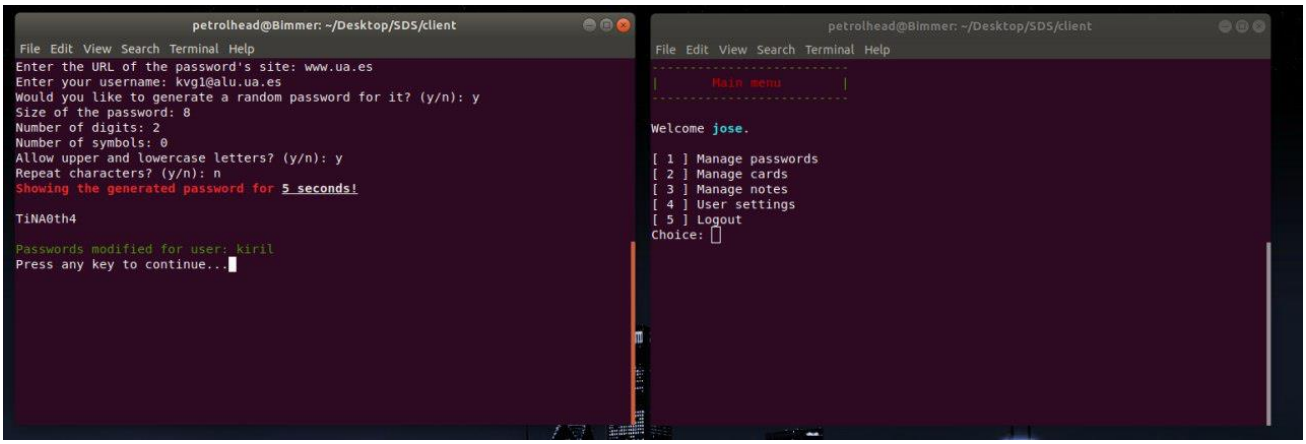
A la izquierda aparece el menú del usuario “kiril” y a la derecha el menú del usuario “jose”. Ambos usuarios se ha acaban de registrar.



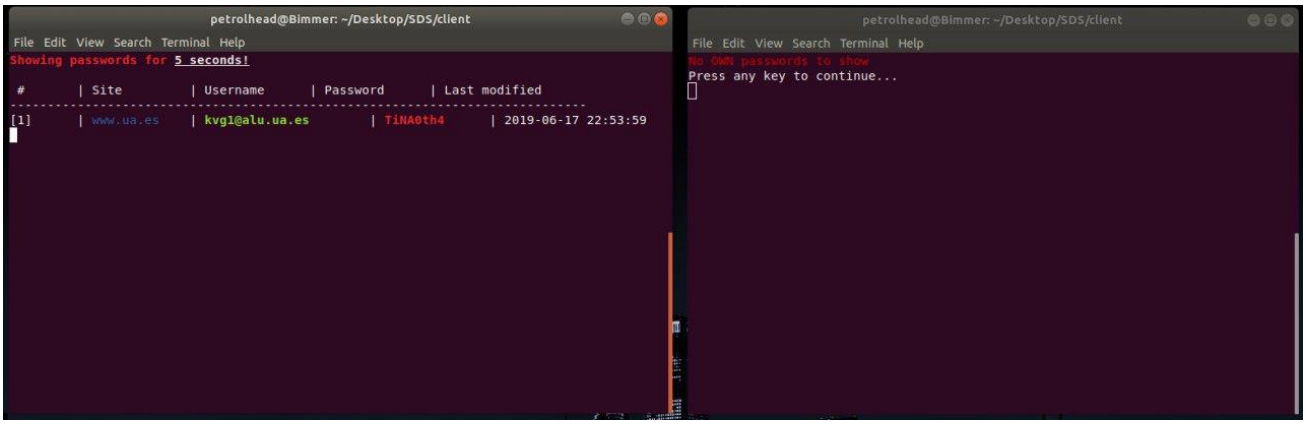
A la izquierda se ha seleccionado la primera opción del menú (Manage passwords) y se procederá a crear una contraseña nueva.



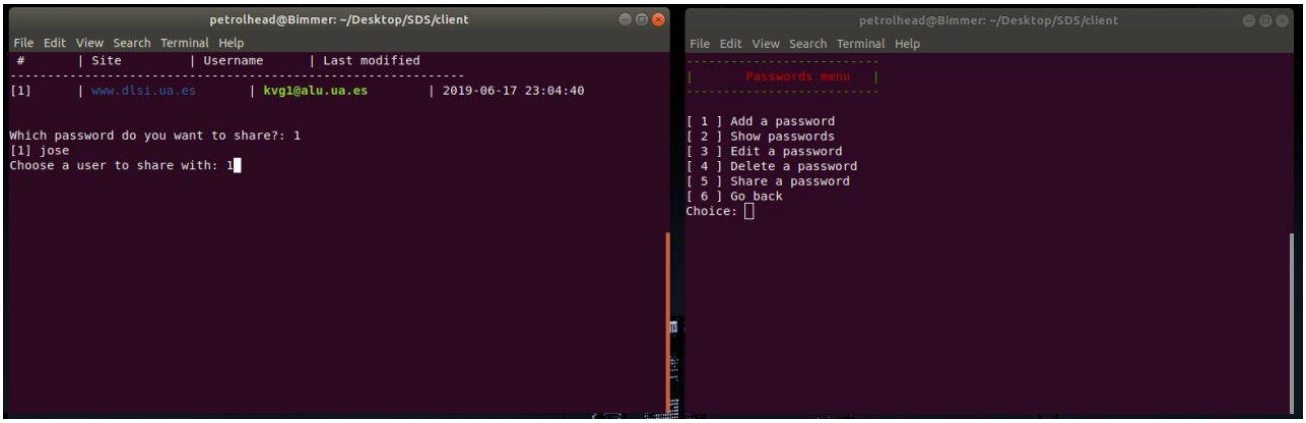
Tras rellenar todos los datos del formulario, se procede a insertar la estructura de la contraseña nueva encriptada en la base de datos sin que el servidor pueda leer los datos.



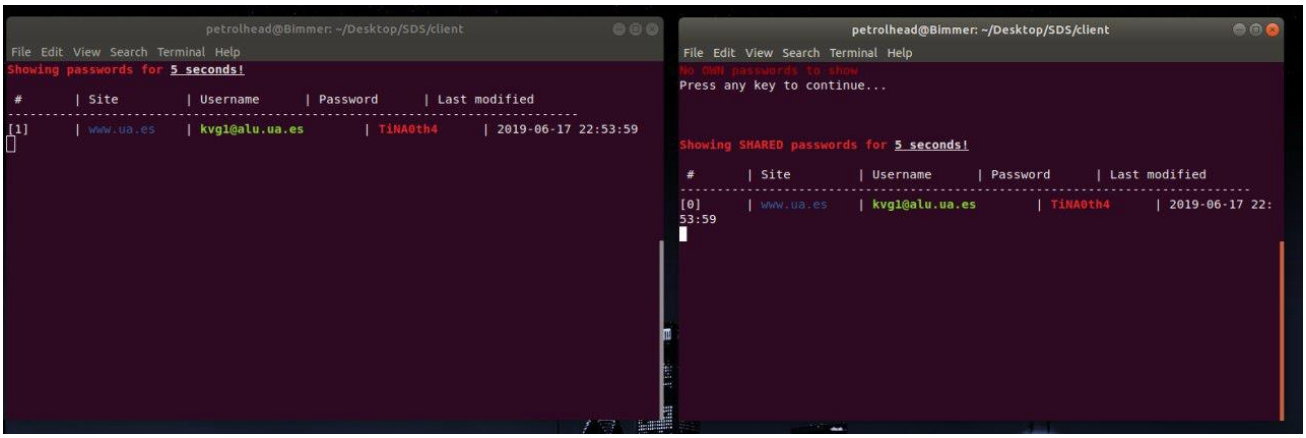
Se muestra la contraseña generada.



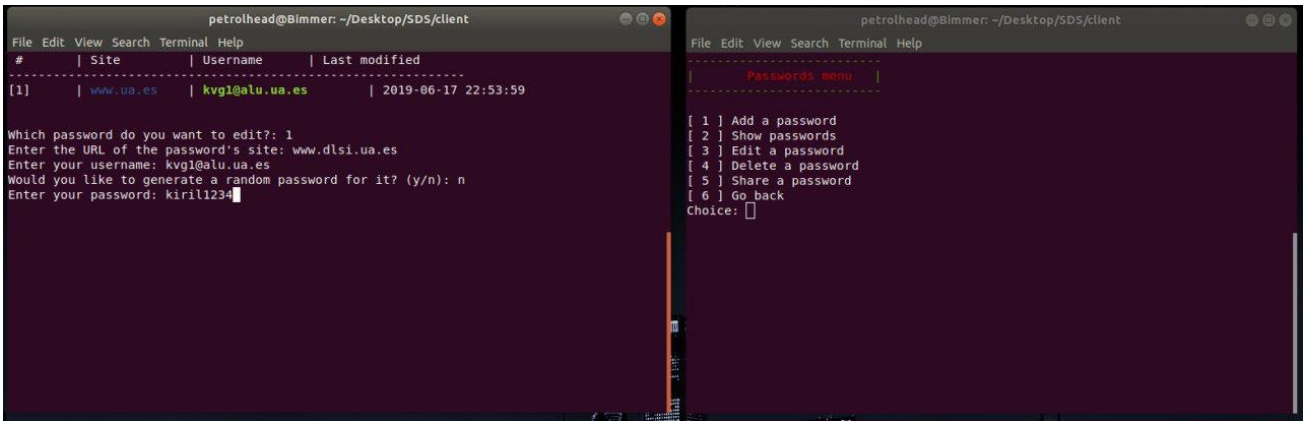
Se procede a seleccionar la opción 5 del menú de contraseñas (Share password) para compartir la contraseña con otros usuarios.



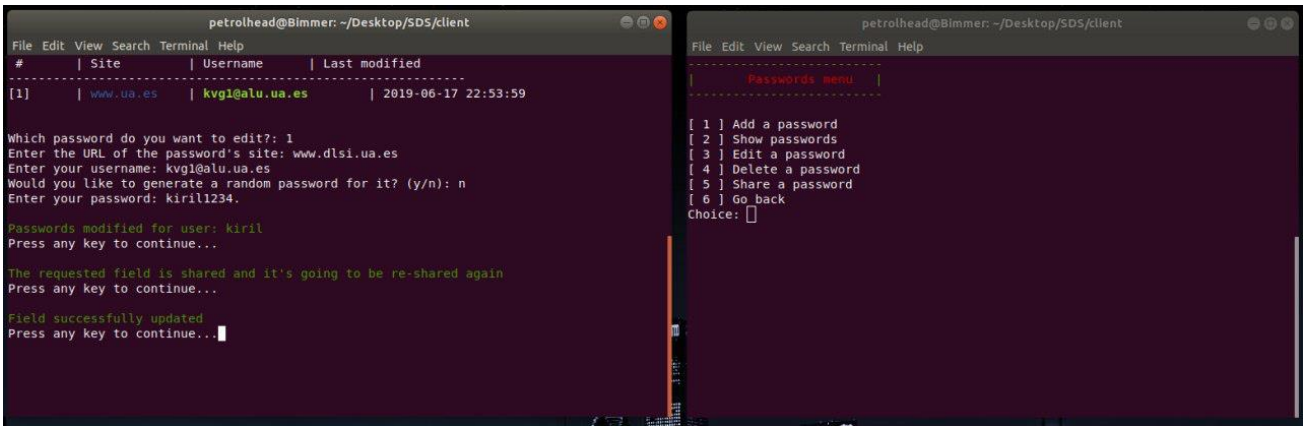
En la siguiente pantalla se muestra, a la izquierda, la contraseña desde la interfaz del usuario “kiril” (propietario) y, a la derecha, se muestra la contraseña compartida con el usuario “jose”.



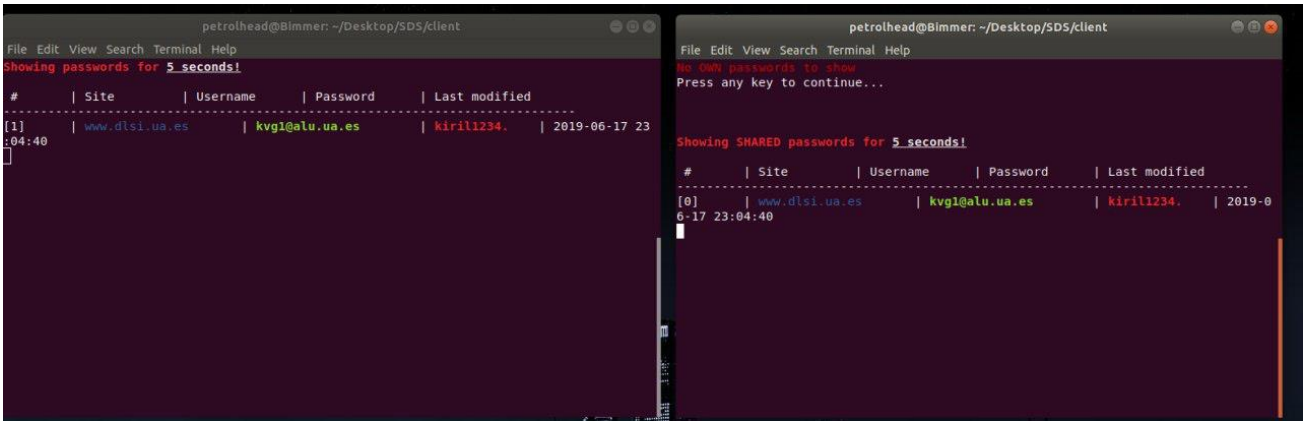
Si el usuario “kiril” procede a editar esta contraseña ya compartida, debe seleccionar la opción 3 del menú de contraseñas (Edit password) y rellenar todos los campos solicitados.



Una vez hecho esto, el programa cliente detectará que la contraseña se está compartiendo con otro/s usuarios mediante la consulta correspondiente a la base de datos y procederá a actualizar la columna con los datos cifrados de la estructura de la contraseña.

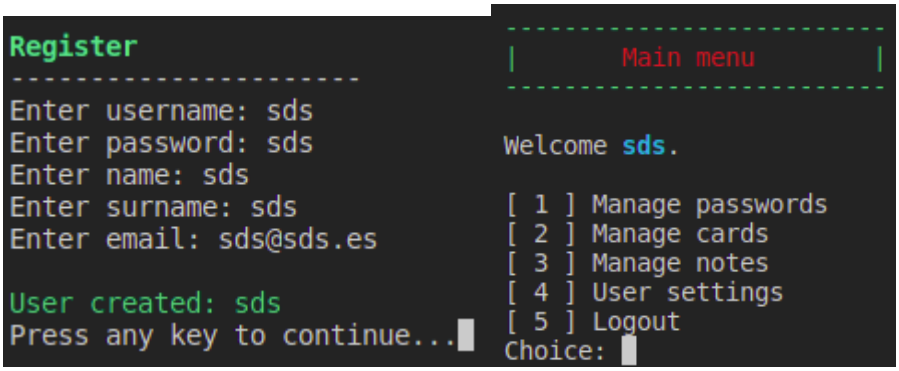


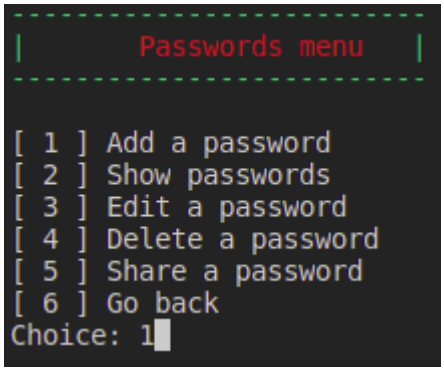
Y, por último, el resultado final tras crear una contraseña, compartirla con otro usuario y editarla después de haberla compartido.

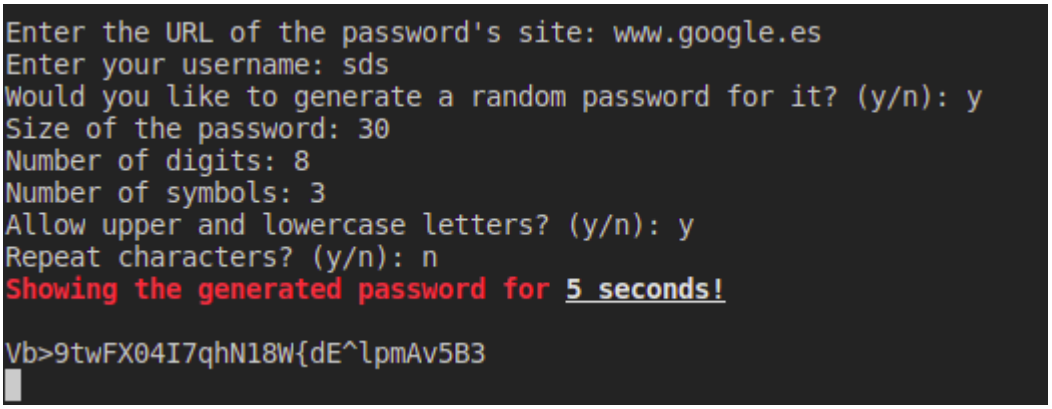


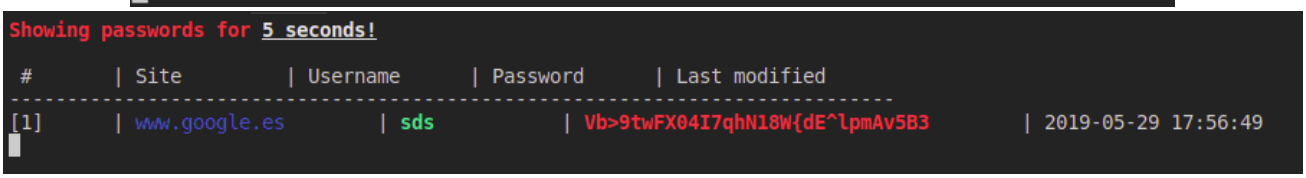
- **EJEMPLO DE USO**

Mediante imágenes mostramos un pequeño ejemplo de uso de la aplicación. Realizaremos este flujo, primero nos registramos, y en cuanto nos registremos la aplicación inicia sesión automáticamente, una vez dentro crearemos una nueva contraseña para nuestro usuario y la URL Google.es, la contraseña la crearemos aleatoriamente. Veremos que la contraseña se muestra 5 segundos y después desaparece. Después listamos nuestras contraseñas y por último cerraremos sesión.









4. Conclusiones

Después de realizar este proyecto nos hemos dado cuenta de que es muy importante planificar la seguridad desde el primer momento y no una vez terminada la implementación ya que por ejemplo el conocimiento cero nos ha supuesto alguna dificultad llevarlo a cabo debido a que no se había tenido en cuenta en el principio y hemos tenido que modificar bastantes líneas de código (todo lo que había en el servidor y CRUD).

Consideramos que la seguridad como medida preventiva es un valor añadido que puede aportar fiabilidad, resiliencia y recuperabilidad. Además de que una mayor seguridad aumenta los beneficios debido a que se reducen los costes asociados a reparación de defectos, a la mejora de la imagen en caso de ataque y es más seguro frente a ataques comunes

5. Despliegue

Le hemos facilitado un usuario y contraseña para que pueda ver la base de datos, estos son los datos:

PONER DATOS

Pasos previos para poder arrancar la aplicación:

Instalar crypto:

go get -u golang.org/x/crypto/scrypt

Instalar mysql:

go get -u github.com/go-sql-driver/mysql

Instalar go-password:

go get -u github.com/sethvargo/go-password/password

Instalar aurora:

go get -u github.com/logrusorgru/aurora

Finalmente, para arrancar el servidor, situarse en la carpeta de este y ejecutar:

*go run *.go [Puerto del servidor]*

Para arrancar el cliente, situarse en la carpeta de este y ejecutar:

*go run *.go [IP del servidor] [Puerto del servidor]*