I.E.S PUNTA DEL VERDE

ADMINISTRACIÓN DE SISTEMAS INFORMÁTICOS EN RED

“INTERFAZ GRÁFICA PARA GESTIONAR USUARIOS Y GRUPOS CON SISTEMA DE LOGS PARA AUDITORÍA”

TFG

Que para obtener el título de:

TÉCNICO EN ADMINISTRACIÓN DE SISTEMAS INFORMÁTICOS EN RED

Presenta:

RUBÉN HANS RODRÍGUEZ

Sevilla, España Junio, 2025

ÍNDICE

CAPÍTULO 1: INTRODUCCIÓN

1.1. INTRODUCCIÓN DEL PROYECTO

En entornos educativos y empresariales, la gestión de usuarios en sistemas Linux es una tarea habitual para los ***administradores de sistemas***. No obstante, para técnicos sin experiencia avanzada en terminal, utilizar comandos como useradd, usermod o deluser puede resultar complejo para algunas personas y es algo ***propenso a errores***.

Este proyecto tiene como objetivo simplificar esa tarea mediante una aplicación gráfica desarrollada en ***Python utilizando Tkinter***. La herramienta permitirá realizar operaciones de gestión de usuarios de manera ***visual, intuitiva y segura.***

Esta herramienta permite ***crear, modificar y eliminar usuarios, gestionar grupos y permisos*** de forma sencilla, y registrar todas las acciones en un sistema de logs para auditoría.

Entre sus ***principales ventajas*** se encuentran la minimización de errores al evitar el uso manual de comandos, el ahorro de tiempo en tareas repetitivas, la accesibilidad para usuarios sin experiencia en terminal y la posibilidad de contar con trazabilidad y control gracias a registros detallados.

1.2. PROPÓSITO

El objetivo es desarrollar una ***aplicación con interfaz gráfica*** que automatice la gestión de usuarios y grupos, permita ejecutar comandos Linux de forma validada y segura, genere ***registros de auditoría en la ruta /var/log/user_manager/*** y facilite la asignación de permisos y grupos mediante menús visuales.

Esta herramienta está dirigida a administradores de sistemas con poca experiencia en línea de comandos, personal de soporte técnico en centros educativos o empresas, y estudiantes de informática que requieran una ***solución didáctica y práctica.***

1.3. OBJETIVOS DEL PROYECTO

El ***objetivo general*** del proyecto es crear una aplicación gráfica que simplifique la gestión de usuarios en ***sistemas Linux***.

Para lograrlo, he planteado varios ***objetivos específicos***: analizar los requisitos técnicos y funcionales, diseñar una ***interfaz intuitiva*** utilizando ***Tkinter***, implementar funciones para ***crear, modificar y eliminar*** usuarios, gestionar grupos y asignaciones de permisos, incluir ***validaciones*** que prevengan errores comunes, integrar un ***sistema de logs*** que registre toda la actividad, y documentar el desarrollo mediante manuales de instalación y uso.

1.4.-COSTE DEL PROYECTO

1.4.1.-COSTES DE DESARROLLO

CONCEPTO	COSTE (€)
HARDWARE	0€
SOFTWARE	0€
PERSONAL (YO)	0€
TOTAL, DE DESARROLLO	0€

Hardware 0€ porque tengo ordenador ya disponible y al igual con el Software dado que Python y Tkinter son gratuitos.

1.4.2 COSTES DE IMPLANTACIÓN

CONCEPTO	COSTE (€)	COMENTARIOS
INSTALACIÓN	0€	(No requiere infraestructura adicional)
FORMACIÓN	100€	(Sesión de 2 horas para administradores)

Podríamos sacar, una hora pagada a 50€ por formarse en la app.

CAPÍTULO 2: ANÁLISIS DEL SISTEMA

2.1.-INTRODUCCIÓN

2.2.-ANÁLISIS DE REQUISITOS

El desarrollo de esta aplicación parte de la necesidad de ***simplificar la gestión de usuarios y grupos*** en sistemas Linux, mediante una ***interfaz gráfica***.

Para ello, presento los siguientes requisitos:

2.2.1.- REQUERIMIENTOS DE HARDWARE

Para el desarrollo y ejecución del proyecto he utilizado el siguiente hardware:

- Ordenador portátil con procesador Intel i5 de 8ª generación.
- Disco duro: 256 GB SSD.
- Memoria RAM de 8 GB.
- 100 MB de espacio en disco para la app.
- Pantalla con resolución mínima de 1366x768
- SO: Ubuntu/Debian (o cualquier distribución Linux).

Este tipo de configuración me ha resultado suficiente tanto para desarrollar como para ejecutar la aplicación ***sin problemas de rendimiento***.

2.2.2.- REQUERIMIENTOS DE SOFTWARE

- SO: Ubuntu/Debian (o cualquier distribución Linux).
- Python 3.10+ y python3-tk.
- Bibliotecas: subprocess, logging, os.

- Permisos de superusuario para la ejecución de comandos administrativos del sistema.
- Editor de código (Opcional): En mi caso utilizo VS Code por comodidad.

2.2.3.-REQUISITOS DE RED

No aplica dado que el sistema ***funciona localmente***. Sin embargo, si se quisiera implementar en ***entornos multiusuario o en red local***, podría adaptarse para trabajar con usuarios remotos o integrar funcionalidades de red en el futuro.

2.2.4.-OTROS REQUISITOS DE INTERÉS

El usuario que ejecute la aplicación debe tener ***permisos de superusuario***, ya que las operaciones de creación, modificación o eliminación de usuarios requieren ***autorización administrativa***.

También cabe destacar que el directorio ***/var/log/user_manager/*** debe existir o poder ser creado con permisos de escritura, ya que en él se almacenan los registros de actividad.

Por último, la GUI, está diseñada para tener compatibilidad con ***resoluciones estándar de pantalla***.

CAPÍTULO 3: ESTUDIO DE LA INFORMACIÓN RELEVANTE

3.1.-INTRODUCCIÓN.

3.2.- VALORACIÓN Y COMPARACIÓN CON ANTECEDENTES

A diferencia de herramientas tradicionales como Webmin, que, aunque potentes requieren una configuración previa compleja y un ***conocimiento técnico considerable***, y de la gestión mediante terminal, que si bien es eficiente puede resultar poco accesible ***para usuarios sin experiencia***.

La GUI que he desarrollado ofrece una ***solución visual, intuitiva y más fácil de usar***, ideal para entornos educativos o profesionales donde se busca simplificar la administración y seguir siendo ***eficientes en su labor***.

3.3.-VALORACIÓN Y COMPARACIÓN CON LOS ACTUALES

La app que he desarrollado se distingue por ***su sencillez, portabilidad y facilidad*** de uso.
Por otro lado, tampoco requiere una ***instalación compleja ni dependencias externas*** y además presenta ***compatibilidad garantizada*** con la integración de los comandos nativos.

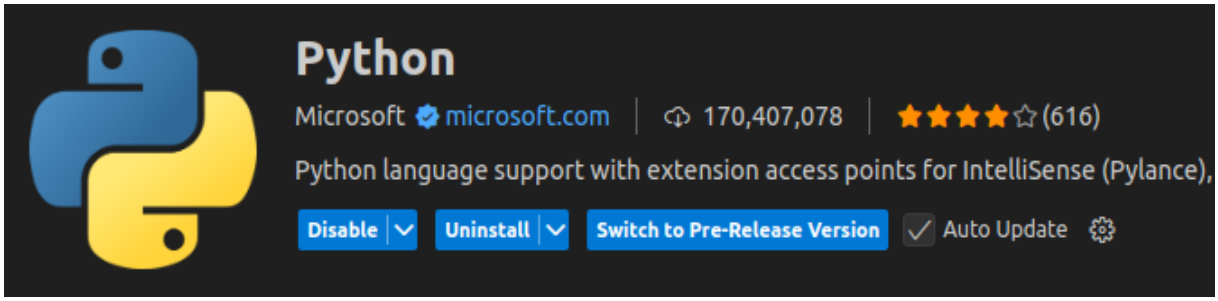
CAPÍTULO 4: TÉCNICAS UTILIZADAS

4.1.-INTRODUCCIÓN.

Como primer paso, lo primero que hice es crear un directorio y dar permisos de ejecución al archivo para trabajar sobre él e ir cambiándolo conforme a mis necesidades.

```
rhr@rhr-VirtualBox:~$ mkdir gestion_usuarios_linux
rhr@rhr-VirtualBox:~$ cd gestion_usuarios_linux
rhr@rhr-VirtualBox:~/gestion_usuarios_linux$ touch user_manager.py
rhr@rhr-VirtualBox:~/gestion_usuarios_linux$ chmod +x user_manager.py
rhr@rhr-VirtualBox:~/gestion_usuarios_linux$
```

Ya que el proyecto está desarrollado en ***Python***, instalé su extensión en ***Visual Code Studio*** para trabajar más eficaz y visualmente.



4.2.-DESARROLLO DE LAS TÉCNICAS.

Una vez instalado Python en VSCode, procedí a importar las cinco bibliotecas necesarias.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Gestión de Usuarios en Linux - GUI
Autor: [Rubén Hans Rodríguez]
Fecha: 31/3/2025
IES Punta del Verde - Dpto. de Informática
"""
import tkinter as tk
from tkinter import ttk, messagebox, filedialog
import subprocess
import logging
import os
from datetime import datetime
```

1. **tkinter** y **ttk** para crear la interfaz gráfica.
2. **subprocess** para ejecutar comandos del sistema.
3. **os** para manipular el sistema de archivos y permisos.
4. **logging** para guardar logs.
5. **datetime** para registrar fechas.

Proseguí con la creación de la clase ***"UserManagerApp"*** que contendrá toda la aplicación.

```
class UserManagerApp:
    # Define una clase para gestionar la interfaz de la aplicación

    def __init__(self, root):
        # Método que se ejecuta al crear una instancia de la clase
        self.root = root  # Guarda la ventana principal (objeto Tk) como atributo
        self.root.title("Gestión de Usuarios Linux - IES Punta del Verde")  # Establece el título de la ventana
        self.root.geometry("800x600")  # Define el tamaño inicial de la ventana

        self.setup_logging()  # Configura el sistema de registro (logs)
        self.create_status_bar()  # Crea la barra de estado en la parte inferior de la ventana
        self.create_menu()  # Crea la barra de menú superior con opciones como "Archivo" y "Ayuda"
        self.create_widgets()  # Crea y organiza los elementos gráficos principales (pestañas y formularios)
```

De forma seguida configuré el ***sistema de logging*** en Python para mi aplicación, por lo que creé un ***logger principal*** que registraba mensajes desde el ***nivel INFO*** y enviaba esos registros a un archivo llamado ***user_manager.log*** dentro de ese directorio. También definí un formato para los mensajes con fecha, nivel y contenido, y registré un mensaje inicial indicando que la aplicación había empezado.

```python
def setup_logging(self):
    """Configura el sistema de logging (compatible con Python 3.8)"""

log_dir = "/var/log/user_manager"  # Ruta donde se almacenarán los archivos de log

if not os.path.exists(log_dir):  # Si el directorio no existe, se crea
    os.makedirs(log_dir, exist_ok=True)  # Crea el directorio de logs si no existe
    os.chmod(log_dir, 0o755)  # Establece permisos (lectura/ejecución para todos, escritura solo para propietario)

logger = logging.getLogger()  # Obtiene el logger principal del sistema de logging
logger.setLevel(logging.INFO)  # Establece el nivel mínimo de mensajes que se van a registrar (INFO o superior)

handler = logging.FileHandler(  # Crea un manejador que escribirá los logs en un archivo
    filename=f"{log_dir}/user_manager.log",  # Ruta y nombre del archivo de log
    encoding='utf-8'  # Codificación para caracteres especiales
)

formatter = logging.Formatter(  # Define el formato de cada mensaje de log
    '%(asctime)s - %(levelname)s - %(message)s',  # Incluirá fecha, nivel de log y mensaje
    datefmt='%d/%m/%Y %H:%M:%S'  # Formato de fecha y hora personalizado
)

handler.setFormatter(formatter)  # Asigna el formato al manejador
logger.addHandler(handler)  # Añade el manejador al logger principal

logging.info("Aplicación iniciada")  # Registra un mensaje indicando que la aplicación ha sido iniciada
```

Creé una ***barra de menú***, con dos menús principales: "***Archivo***", donde puse opciones para exportar logs y salir de la aplicación, y "***Ayuda***", con acceso al manual de usuario y a la información "***Acerca de***". Finalmente, asigné esa barra de menú a la ventana principal.

```python
def create_menu(self):
    """Crea la barra de menú"""
    menubar = tk.Menu(self.root)  # Crea la barra de menú principal asociada a la ventana raíz

    # Menú Archivo
    file_menu = tk.Menu(menubar, tearoff=0)  # Crea un submenú para "Archivo" sin opción de separación flotante
    file_menu.add_command(label="Exportar logs", command=self.export_logs)  # Añade opción para exportar logs, que llama a self.export_l
    file_menu.add_separator()  # Añade una línea separadora en el menú
    file_menu.add_command(label="Salir", command=self.root.quit)  # Añade opción para salir de la aplicación, cierra la ventana raíz
    menubar.add_cascade(label="Archivo", menu=file_menu)  # Añade el submenú "Archivo" a la barra de menú principal

    # Menú Ayuda
    help_menu = tk.Menu(menubar, tearoff=0)  # Crea un submenú para "Ayuda" sin opción de separación flotante
    help_menu.add_command(label="Manual de usuario", command=self.show_help)  # Añade opción para mostrar el manual de usuario
    help_menu.add_command(label="Acerca de", command=self.show_about)  # Añade opción para mostrar información "Acerca de"
    menubar.add_cascade(label="Ayuda", menu=help_menu)  # Añade el submenú "Ayuda" a la barra de menú principal

    self.root.config(menu=menubar)  # Configura la ventana raíz para que use esta barra de menú creada
```

Seguí haciendo un ***widget tipo notebook*** en la ventana principal y añadí tres pestañas: una para gestionar usuarios, otra para gestionar grupos y una última para mostrar el registro de actividades. Para cada pestaña llamé a funciones que configuraban su contenido específico, y finalmente empaqué el widget para que se ajustara bien dentro de la ventana.

```python
def create_widgets(self):
    """Crea los widgets principales"""
    self.tab_control = ttk.Notebook(self.root)  # Creo un widget tipo pestañas (Notebook) asociado a la ventana raíz

    # Pestaña de usuarios
    self.tab_user = ttk.Frame(self.tab_control)  # Hago un frame para la pestaña de usuarios dentro del notebook
    self.tab_control.add(self.tab_user, text='Gestión de Usuarios')  # Añado la pestaña al notebook con su título
    self.create_user_tab()  # Llamo a la función que configura el contenido de la pestaña de usuarios

    # Pestaña de grupos
    self.tab_group = ttk.Frame(self.tab_control)  #frame para la pestaña de grupos
    self.tab_control.add(self.tab_group, text='Gestión de Grupos')  # Añado la pestaña de grupos al notebook
    self.create_group_tab()  # Configuro el contenido de la pestaña de grupos

    # Pestaña de logs
    self.tab_logs = ttk.Frame(self.tab_control)  # Creo un frame para la pestaña de logs
    self.tab_control.add(self.tab_logs, text='Registro de Actividades')  # Añado la pestaña de logs al notebook
    self.create_logs_tab()  #  contenido de la pestaña de logs

    self.tab_control.pack(expand=1, fill="both", padx=5, pady=5)  #  para que se expanda y rellene el espacio disponible con márgenes
```

Para la pestaña de ***gestión de usuarios***, la hice con un formulario para ingresar y ***modificar datos*** como nombre, contraseña, directorio home, shell y grupos del usuario. Luego cargué los grupos del sistema en una lista para que se pudieran seleccionar múltiples opciones añadiendo botones para ***crear, modificar, eliminar y limpiar usuarios***. Para terminar con este trozo de código, configuré una tabla con scroll para ***mostrar los usuarios del sistema***.

```python
def create_user_tab(self):
    """Crea la pestaña de gestión de usuarios"""
main_frame = ttk.Frame(self.tab_user)  # Creo un contenedor principal dentro de la pestaña de usuarios
main_frame.pack(fill="both", expand=True, padx=10, pady=10)  # Lo empaqué para que ocupe todo el espacio con margen

form_frame = ttk.LabelFrame(main_frame, text="Formulario de Usuario", padding="10")  # Creo un grupo con título para el formulario
form_frame.pack(fill="x", padx=5, pady=5)  # Lo empaqué con margen, ocupando ancho completo

# Campos del formulario
ttk.Label(form_frame, text="Nombre de usuario:").grid(row=0, column=0, sticky="w", pady=2)  # Etiqueta para usuario
self.username_entry = ttk.Entry(form_frame)  # Campo de texto para nombre de usuario
self.username_entry.grid(row=0, column=1, sticky="ew", padx=5, pady=2)  # Lo ubico en la grilla

ttk.Label(form_frame, text="Contraseña:").grid(row=1, column=0, sticky="w", pady=2)  # Etiqueta para contraseña
self.password_entry = ttk.Entry(form_frame, show="*")  # Campo para contraseña oculta
self.password_entry.grid(row=1, column=1, sticky="ew", padx=5, pady=2)

ttk.Label(form_frame, text="Confirmar contraseña:").grid(row=2, column=0, sticky="w", pady=2)  # Confirmación contraseña
self.confirm_password_entry = ttk.Entry(form_frame, show="*")  # Campo para confirmar contraseña oculta
self.confirm_password_entry.grid(row=2, column=1, sticky="ew", padx=5, pady=2)

ttk.Label(form_frame, text="Directorio home:").grid(row=3, column=0, sticky="w", pady=2)  # Etiqueta para home
self.home_entry = ttk.Entry(form_frame)  # Campo para ruta home
self.home_entry.grid(row=3, column=1, sticky="ew", padx=5, pady=2)
self.home_entry.insert(0, "/home/")  # Pongo valor por defecto

ttk.Label(form_frame, text="Shell:").grid(row=4, column=0, sticky="w", pady=2)  # Etiqueta para shell
self.shell_combobox = ttk.Combobox(form_frame, values=[  # Combo con opciones de shells disponibles
    "/bin/bash",
    "/bin/sh",
    "/usr/sbin/nologin",
    "/bin/zsh",
    "/bin/fish"
])
```

```
    self.shell_combobox.grid(row=4, column=1, sticky="ew", padx=5, pady=2)
    self.shell_combobox.set("/bin/bash")  # Opción por defecto

    # Lista de grupos
    ttk.Label(form_frame, text="Grupos:").grid(row=5, column=0, sticky="nw", pady=2)  # Etiqueta para grupos
    self.group_listbox = tk.Listbox(form_frame, selectmode="multiple", height=6, exportselection=False)  # Lista múltiple para selecciona
    self.group_listbox.grid(row=5, column=1, sticky="ew", padx=5, pady=2)

    # Cargar grupos del sistema
    groups_result = subprocess.run(['getent', 'group'], capture_output=True, text=True, encoding='utf-8')  # Ejecuto comando para obtener
    if groups_result.returncode == 0:
        for line in groups_result.stdout.splitlines():  # Para cada grupo
            group = line.split(":")[0]  # Extraigo el nombre del grupo
            self.group_listbox.insert(tk.END, group)  # Lo añado a la lista

    button_frame = ttk.Frame(form_frame)  # Creo un contenedor para botones
    button_frame.grid(row=6, column=0, columnspan=2, pady=10)  # Lo ubico en la grilla

    # Botones para crear, modificar, eliminar y limpiar campos
    ttk.Button(button_frame, text="Crear Usuario", command=self.create_user).grid(row=0, column=0, padx=5)
    ttk.Button(button_frame, text="Modificar Usuario", command=self.modify_user).grid(row=0, column=1, padx=5)
    ttk.Button(button_frame, text="Eliminar Usuario", command=self.delete_user).grid(row=0, column=2, padx=5)
    ttk.Button(button_frame, text="Limpiar Campos", command=self.clear_fields).grid(row=0, column=3, padx=5)

    # Lista de usuarios
    list_frame = ttk.LabelFrame(main_frame, text="Usuarios del Sistema", padding="10")  # Grupo para mostrar usuarios
    list_frame.pack(fill="both", expand=True, padx=5, pady=5)  # El empaquetado

    self.user_tree = ttk.Treeview(list_frame, columns=("username", "uid", "gid", "groups", "home", "shell"), show="headings")  # Tabla pa

    for col in ("username", "uid", "gid", "groups", "home", "shell"):  # Configuro columnas
        self.user_tree.heading(col, text=col.capitalize())  # Título de columna
        self.user_tree.column(col, width=100, anchor="w")  # Ancho y alineación

    scrollbar = ttk.Scrollbar(list_frame, orient="vertical", command=self.user_tree.yview)  # Barra de desplazamiento vertical
    self.user_tree.configure(yscrollcommand=scrollbar.set)  # Asocio la barra
```

De igual forma que hice la tabla de usuarios, también necesitaba una tabla de grupos así que creé la pestaña de ***gestión de grupos*** con un formulario para ingresar el nombre y el ID del grupo y ya pude añadir botones para ***crear, eliminar y limpiar los campos del formulario***. También configuré una ***tabla con scroll*** para mostrar los grupos del sistema y cargué esa información al iniciarla e hice que se detecte cuando se ***selecciona un grupo*** para trabajar con él.

```
def create_group_tab(self):
    """Crea la pestaña de gestión de grupos"""
    main_frame = ttk.Frame(self.tab_group)  # Creo un contenedor principal dentro de la pestaña de grupos
    main_frame.pack(fill="both", expand=True, padx=10, pady=10)  # Lo empaqué para que ocupe todo el espacio con margen

    form_frame = ttk.LabelFrame(main_frame, text="Formulario de Grupo", padding="10")  # Creo un grupo con título para el formulario
    form_frame.pack(fill="x", padx=5, pady=5)  # Lo empaqué para que ocupe el ancho disponible con margen

    ttk.Label(form_frame, text="Nombre del grupo:").grid(row=0, column=0, sticky="w", pady=2)  # Etiqueta para nombre de grupo
    self.group_entry = ttk.Entry(form_frame)  # Campo de texto para ingresar nombre de grupo
    self.group_entry.grid(row=0, column=1, sticky="ew", padx=5, pady=2)

    ttk.Label(form_frame, text="ID del grupo (GID):").grid(row=1, column=0, sticky="w", pady=2)  # Etiqueta para GID
    self.gid_entry = ttk.Entry(form_frame)  # Campo de texto para ingresar GID
    self.gid_entry.grid(row=1, column=1, sticky="ew", padx=5, pady=2)

    button_frame = ttk.Frame(form_frame)  # Creo un contenedor para botones
    button_frame.grid(row=2, column=0, columnspan=2, pady=10)  # Lo ubico en la grilla abarcando dos columnas

    # Botones para crear, eliminar y limpiar campos de grupo
    ttk.Button(button_frame, text="Crear Grupo", command=self.create_group).grid(row=0, column=0, padx=5)
    ttk.Button(button_frame, text="Eliminar Grupo", command=self.delete_group).grid(row=0, column=1, padx=5)
    ttk.Button(button_frame, text="Limpiar Campos", command=self.clear_group_fields).grid(row=0, column=2, padx=5)

    # Lista de grupos
    list_frame = ttk.LabelFrame(main_frame, text="Grupos del Sistema", padding="10")  # Grupo para mostrar grupos existentes
    list_frame.pack(fill="both", expand=True, padx=5, pady=5)  # Lo empaqué para que ocupe todo el espacio con margen

    self.group_tree = ttk.Treeview(list_frame, columns=("groupname", "gid", "members"), show="headings")  # Tabla para mostrar grupos

    for col in ("groupname", "gid", "members"):  # Configuro las columnas de la tabla
        self.group_tree.heading(col, text=col.capitalize())  # Pongo los títulos de columna
        self.group_tree.column(col, width=100, anchor="w")  # Defino ancho y alineación

    scrollbar = ttk.Scrollbar(list_frame, orient="vertical", command=self.group_tree.yview)  # Barra de desplazamiento vertical
    self.group_tree.configure(yscrollcommand=scrollbar.set)  # Asocio la barra al árbol
    self.group_tree.pack(side="left", fill="both", expand=True)  # Empaqué la tabla para ocupar el espacio disponible
    scrollbar.pack(side="right", fill="y")  # Empaqué la barra a la derecha

    self.load_group_list()  # Cargo la lista de grupos en la tabla
    self.group_tree.bind("<<TreeviewSelect>>", self.on_group_select)  # Asocio evento para selección de grupo
```

Para finalizar las tablas, hice la pestaña para visualizar los logs del sistema donde añadí un área de texto con scroll para mostrar los registros y dos botones: uno para ***actualizarlos*** y otro para ***limpiarlos***. Al finalizar, se cargan automáticamente los logs existentes cuando abrimos la pestaña.

```
def create_logs_tab(self):
 """Crea la pestaña de visualización de logs"""
main_frame = ttk.Frame(self.tab_logs)  # Creé un marco principal dentro de la pestaña de logs
main_frame.pack(fill="both", expand=True, padx=10, pady=10)  # Lo empaqué para que ocupe todo el espacio con márgenes

self.log_text = tk.Text(main_frame, wrap="word", state="disabled")  # Wdget de texto solo lectura para mostrar los logs
scrollbar = ttk.Scrollbar(main_frame, command=self.log_text.yview)  # Bbarra de desplazamiento vertical
self.log_text.configure(yscrollcommand=scrollbar.set)  # Asocié el scroll al widget de texto

self.log_text.pack(side="left", fill="both", expand=True)  # Puse el área de texto al lado izquierdo
scrollbar.pack(side="right", fill="y")  # la barra al lado derecho

button_frame = ttk.Frame(main_frame)  #Contenedor para los botones debajo del área de texto
button_frame.pack(fill="x", pady=5)  # Lo puse para que ocupe el ancho completo

ttk.Button(button_frame, text="Actualizar Logs", command=self.update_logs).pack(side="left", padx=5)  # Botón para actualizar logs
ttk.Button(button_frame, text="Limpiar Logs", command=self.clear_logs).pack(side="left", padx=5)  # Botón para limpiar logs

self.update_logs()  # Llamé a la función para cargar los logs al iniciar la pestaña
```

Por último, respecto a este apartado, elaboré una ***barra de estado*** en la parte inferior de la ventana principal usando una etiqueta con texto dinámico. Luego, inicialicé el mensaje con "***Listo***" y programé una función para poder actualizar ese texto en tiempo real durante la ejecución de la GUI.

```
def create_status_bar(self):
    """Crea la barra de estado"""
self.status_var = tk.StringVar()  # Variable para manejar el texto de la barra de estado
self.status_var.set("Listo")  # Inicialicé la barra con el texto "Listo"

status_bar = ttk.Label(  # Etiqueta para usar como barra de estado
    self.root,
    textvariable=self.status_var,  #Etiqueta asociada a la variable de estado
    relief="sunken",  # Establecí un relieve hundido para que parezca una barra de estado
    anchor="w"  # texto a la izquierda
)

status_bar.pack(side="bottom", fill="x")  # Coloqué la barra en la parte inferior y la hice ocupar todo el ancho


def update_status(self, message):
    """Actualiza el mensaje en la barra de estado"""
    self.status_var.set(message)  # Cambié el texto de la barra al mensaje recibido
self.root.update_idletasks()  # Forcé la actualización de la interfaz para que el cambio se muestre de inmediato
```

4.2.1-FUNCIONALIDADES DEL USUARIO

Empezando con las ***funcionalidades del usuario***, cargué la lista de usuarios del sistema ejecutando el comando getent passwd donde procesé cada línea para extraer los datos relevantes como ***nombre, UID, GID, directorio home y shell***. Aquí consulté también los grupos de cada usuario y mostré esta información en el árbol de usuarios de la interfaz. Por último, limpié la lista

anterior antes de mostrar la nueva y gestioné errores y excepciones para informar al usuario y registrar los ***fallos en el log***.

```python
def create_group_tab(self):
    def load_user_list(self):
        """Carga la lista de usuarios del sistema"""
    try:
        for item in self.user_tree.get_children():
            self.user_tree.delete(item)  # Limpié todos los elementos previos del árbol de usuarios

        result = subprocess.run(['getent', 'passwd'], capture_output=True, text=True, encoding='utf-8')  # Ejecuté el comando para obtener la lista de usuarios

        if result.returncode == 0:
            users = result.stdout.splitlines()  # Separé cada usuario en una línea

            for user in users:
                parts = user.split(':')  # Separé los campos de cada línea
                if len(parts) >= 7:
                    username = parts[0]
                    uid = parts[2]
                    gid = parts[3]
                    home = parts[5]
                    shell = parts[6]

                    # Obtuve los grupos a los que pertenece el usuario
                    groups_result = subprocess.run(
                        ['groups', username],
                        capture_output=True,
                        text=True,
                        encoding='utf-8'
                    )

                    groups = groups_result.stdout.split(':')[1].strip() if groups_result.returncode == 0 else ""

                    # Inserté el usuario en el árbol de visualización
                    self.user_tree.insert("", "end", values=(username, uid, gid, groups, home, shell))

            self.update_status(f"Usuarios cargados: {len(users)}")  # Actualicé la barra de estado
            logging.info("Lista de usuarios actualizada")  # Registré en el log que se cargaron los usuarios

        else:
            messagebox.showerror("Error", "Error al obtener usuarios")  # Mostré un mensaje si falló el comando
            logging.error(f"Error getent: {result.stderr}")  # Registré el error en el log

    except Exception as e:
        messagebox.showerror("Error", f"Excepción: {str(e)}")  # Mostré cualquier otra excepción
        logging.error(f"Excepción load_user_list: {str(e)}")  # Registré la excepción en el log
```

Seguidamente programé el evento para que, al ***seleccionar un usuario***, los datos se cargaran automáticamente en el formulario (nombre, directorio home, shell y grupos). Más tarde, rellené los nuevos con la información del usuario que seleccionamos, marcando también sus ***grupos correspondientes*** en la lista.

```python
def on_user_select(self, event):
    """Evento al seleccionar un usuario"""
    selected = self.user_tree.selection()  # Obtuve el elemento seleccionado del árbol
    if selected:
        user_data = self.user_tree.item(selected[0])['values']  # Extraje los valores del usuario seleccionado

        self.username_entry.delete(0, tk.END)  # Campo de nombre de usuario
        self.username_entry.insert(0, user_data[0])  # Nombre del usuario seleccionado

        self.home_entry.delete(0, tk.END)  # Campo del directorio home
        self.home_entry.insert(0, user_data[4])  #Valor del directorio home

        self.shell_combobox.set(user_data[5])  # Establecí el valor del combobox de shell

        self.group_listbox.selection_clear(0, tk.END)  # Desmarqué todos los grupos previamente seleccionados

        # Si hay información de grupos, los selecciona en la lista
        if len(user_data) > 3 and user_data[3]:
            user_groups = user_data[3].split()  # Separé los grupos del usuario en una lista
            all_groups = self.group_listbox.get(0, tk.END)  # Obtuve todos los grupos disponibles
            for idx, group in enumerate(all_groups):
                if group in user_groups:
                    self.group_listbox.selection_set(idx)  # Seleccioné los grupos a los que pertenece el usuario
```

Asimismo, validé los campos del usuario comprobando que el nombre de usuario y el directorio home no estuvieran vacíos de forma que, si se había ingresado una contraseña, verifique que ***coincida con su confirmación*** y que tuviera al menos 8 caracteres. Si algo no estaba bien, mostré advertencias y cancelé el proceso.

```
    def validate_user_fields(self):
        """Valida los campos del formulario de usuario"""
    # Valores ingresados en los campos del formulario
    username = self.username_entry.get().strip()
    password = self.password_entry.get()
    confirm_password = self.confirm_password_entry.get()
    home = self.home_entry.get().strip()

    # Verifica si se ingresó un nombre de usuario
    if not username:
        messagebox.showwarning("Validación", "Nombre de usuario requerido")
        return False

    # Verifica si se ingresó un directorio home
    if not home:
        messagebox.showwarning("Validación", "Directorio home requerido")
        return False

    # Si se ingresó una contraseña, valida que ambas coincidan y tengan la longitud mínima
    if password or confirm_password:
        if password != confirm_password:
            messagebox.showwarning("Validación", "Las contraseñas no coinciden")
            return False

        if len(password) < 8:
            messagebox.showwarning("Validación", "La contraseña debe tener al menos 8 caracteres")
            return False

    return True  # Si todo está correcto
```

Para la ***creación del usuario*** validé los datos del formulario y, si eran correctos, comprobé que el ***usuario no existiera***. Luego construí y ejecuté el comando useradd con los parámetros necesarios. Si el usuario se ***creó correctamente***, la app lo añade a los grupos seleccionados, configura su contraseña (nos da la opción), lo actualiza en la lista y muestra un mensaje de éxito. En caso de error, salta un mensaje y se registra en los logs:

```
def create_user(self):
    """Crea un nuevo usuario"""
    # Primero validé que los campos del formulario fueran correctos
    if not self.validate_user_fields():
        return
    # Obtuve los datos ingresados por el usuario
    username = self.username_entry.get().strip()
    password = self.password_entry.get()
    home = self.home_entry.get().strip()
    shell = self.shell_combobox.get()
    try:
        # Verifiqué si el usuario ya existía en el sistema
        if self.user_exists(username):
            messagebox.showerror("Error", f"Usuario '{username}' ya existe")
            return
        # Construí el comando para crear el usuario
        cmd = ['sudo', 'useradd']
        if home:
            cmd.extend(['-d', home])      # Especificar directorio home
        if shell:
            cmd.extend(['-s', shell])     # Especificar shell
        cmd.extend(['-m', username])      # Crear también el directorio home
        # Ejecuté el comando
        result = subprocess.run(cmd, capture_output=True, text=True, encoding='utf-8')
        if result.returncode == 0:
            # Si el usuario se creó correctamente, lo añadí a los grupos seleccionados
            selected_indices = self.group_listbox.curselection()
            selected_groups = [self.group_listbox.get(i) for i in selected_indices]
            if selected_groups:
                subprocess.run(['sudo', 'usermod', '-aG', ','.join(selected_groups), username])
            # Si se proporcionó una contraseña, la establecí
            if password:
                self.set_password(username, password)
            # Mostré mensaje de éxito, registré en logs y actualicé la lista de usuarios
            messagebox.showinfo("Éxito", f"Usuario '{username}' creado")
            logging.info(f"Usuario creado: {username}")
            self.load_user_list()
            self.clear_fields()
        else:
            # Si ocurrió un error, lo mostré y lo registré
            messagebox.showerror("Error", f"Error al crear usuario: {result.stderr}")
            logging.error(f"Error create_user: {result.stderr}")
    except Exception as e:
        # Manejé cualquier excepción inesperada
        messagebox.showerror("Error", f"Error: {str(e)}")
        logging.error(f"Excepción create_user: {str(e)}")
```

A la hora de la ***modificación del usuario***, también hice validar los datos del formulario y verificar si el usuario existía. Luego construí el comando usermod con las nuevas configuraciones (como el home o shell), lo ejecuté y, si fue exitoso, actualiza los grupos del usuario y su contraseña en caso necesario. Si hubo errores, se registran en el log.

```
def modify_user(self):
    """Modifica un usuario existente"""
    # Validé los campos del formulario antes de continuar
    if not self.validate_user_fields():
        return
    # Obtuve los valores actuales del formulario
    username = self.username_entry.get().strip()
    password = self.password_entry.get()
    home = self.home_entry.get().strip()
    shell = self.shell_combobox.get()
    try:
        # Verifiqué si el usuario existe antes de modificarlo
        if not self.user_exists(username):
            messagebox.showerror("Error", f"Usuario '{username}' no existe")
            return
        # Comando para modificar el usuario
        cmd = ['sudo', 'usermod']
        if home:
            cmd.extend(['-d', home])  # Cambiar directorio home
        if shell:
            cmd.extend(['-s', shell])  # Cambiar shell
        cmd.append(username)  # Nombre del usuario
        # Ejecución del comando
        result = subprocess.run(cmd, capture_output=True, text=True, encoding='utf-8')
        if result.returncode == 0:
            # Si se modificó exitosamente, actualiza los grupos del usuario
            selected_indices = self.group_listbox.curselection()
            selected_groups = [self.group_listbox.get(i) for i in selected_indices]
            if selected_groups:
                subprocess.run(['sudo', 'usermod', '-G', ','.join(selected_groups), username])
            # Si se proporcionó nueva contraseña, la cambia
            if password:
                self.set_password(username, password)
            #Mensaje de éxito, registré en logs y recargué la lista
            messagebox.showinfo("Éxito", f"Usuario '{username}' actualizado")
            logging.info(f"Usuario modificado: {username}")
            self.load_user_list()
        else:
            # Si hubo error, mostré mensaje y registré en logs
            messagebox.showerror("Error", f"Error al modificar: {result.stderr}")
            logging.error(f"Error modify_user: {result.stderr}")
    except Exception as e:
        # Manejé cualquier excepción inesperada
        messagebox.showerror("Error", f"Error: {str(e)}")
        logging.error(f"Excepción modify_user: {str(e)}")
```

En cuanto a la ***eliminación de usuarios***, validé como en las demás configuraciones que se haya seleccionado un usuario y que ***existiera en el sistema***. Luego pedí confirmación antes de ejecutar userdel -r para ***eliminar al usuario junto con su directorio home***. Si el comando fue exitoso, muestra un mensaje y si hubo errores, puse cuadros de información con advertencias más claras (como buzón inexistente o eliminación manual del home).

```
def delete_user(self):
    """Elimina un usuario con manejo de errores mejorado"""
    # Obtengo el nombre de usuario desde el formulario
    username = self.username_entry.get().strip()
    # Valido que se haya ingresado un nombre de usuario
    if not username:
        messagebox.showwarning("Validación", "Seleccione un usuario")
        return
    # Verifico que el usuario exista antes de intentar eliminarlo
    if not self.user_exists(username):
        messagebox.showerror("Error", "El usuario no existe")
        return
    # Solicito confirmación al usuario antes de eliminar
    confirm = messagebox.askyesno(
        "Confirmar eliminación",
        f"¿Eliminar usuario '{username}'?\n\n" +
        "Notas:\n- Ignore advertencias de buzón\n" +
        "- El home podría requerir eliminación manual",
        icon="warning"
    )
    if not confirm:
        return
    try:
        #Evitar problemas con espacios
        sanitized_user = username.replace(" ", "\\ ")
        # Ejecuté el comando userdel con la opción -r para eliminar home y buzón
        result = subprocess.run(
            ['sudo', 'userdel', '-r', sanitized_user],
            capture_output=True,
            text=True,
            encoding='utf-8',
            errors='replace'  # Manejo de errores de codificación
        )
        if result.returncode == 0:
            # Usuario eliminado con éxito
            msg = f"Usuario '{username}' eliminado"
            messagebox.showinfo("Éxito", msg)
            logging.info(msg)
        else:
            # Si hubo errores, los interpreto y muestro advertencias más amigables
            error_msg = result.stderr
            if "mail spool" in error_msg:
                error_msg = "✓ Buzón no encontrado (acción segura)"
            if "/home/ no pertenece" in error_msg:
                error_msg = f"⚠ Elimine manualmente: sudo rm -rf /home/{username}"

            messagebox.showwarning("Eliminación parcial", f"Usuario desvinculado\n{error_msg}")
            logging.warning(f"Eliminación parcial: {error_msg}")
        # Actualizo la interfaz después de la eliminación
        self.load_user_list()
        self.clear_fields()
    except Exception as e:
        # Manejo de excepciones generales
        error_msg = f"Error crítico: {str(e)}"
        messagebox.showerror("Error", error_msg)
        logging.error(error_msg)
```

Como siguiente paso para los usuarios ya existentes, establezco sus comprobaciones con la el comando id. Si el ***usuario existía***, el comando se ejecutaba sin errores y devolvía ***True***. En caso contrario, se lanzaba una excepción y retornaba **False**.

```python
def user_exists(self, username):
    """Verifica si un usuario existe"""
    try:
        # Ejecuto el comando 'id' con el nombre del usuario
        # Si el usuario existe, el comando devolverá código de retorno 0 (éxito)
        subprocess.run(
            ['id', username],
            check=True,                    # Lanza una excepción si el comando falla
            stdout=subprocess.PIPE,        # Captura la salida estándar
            stderr=subprocess.PIPE,        # Captura la salida de error
            encoding='utf-8'               # Asegura que la salida sea texto y no bytes
        )
        return True  # El usuario existe
    except subprocess.CalledProcessError:
        return False  # El comando falló: el usuario no existe
```

Para ***establecer la contraseña***, uso el comando ***chpasswd*** a través de un proceso hijo con subprocess.Popen. De esta forma el nombre de usuario y la nueva contraseña se envía por la entrada estándar del proceso.

```python
def set_password(self, username, password):
    """Establece la contraseña del usuario"""
    try:
        # Inicio un proceso para ejecutar 'sudo chpasswd', que cambia contraseñas
        process = subprocess.Popen(
            ['sudo', 'chpasswd'],
            stdin=subprocess.PIPE,    # Permito enviar datos al proceso por entrada estándar
            text=True,                # Trabajo con texto en lugar de bytes
            encoding='utf-8'          # Codificación UTF-8 para enviar la contraseña correctamente
        )
        # Envío el usuario y contraseña con formato "usuario:contraseña\n"
        process.communicate(f"{username}:{password}\n")
        # Registro en logs que la contraseña fue actualizada correctamente
        logging.info(f"Contraseña actualizada para {username}")
    except Exception as e:
        # Si ocurre un error, lo registro en los logs y lanzo la excepción
        logging.error(f"Error set_password: {str(e)}")
        raise
```

Por último, para el apartado de las funciones de usuario hice las funciones para que se ***limpien todos los campos*** del formulario de usuario, restaurando valores predeterminados donde corresponda.

```
def clear_fields(self):
    """Limpia los campos del formulario"""
    # Borra el contenido del campo de nombre de usuario
    self.username_entry.delete(0, tk.END)
    # Borra el contenido del campo de contraseña
    self.password_entry.delete(0, tk.END)
    # Borra el contenido del campo de confirmación de contraseña
    self.confirm_password_entry.delete(0, tk.END)
    # Borra el contenido del campo de directorio home
    self.home_entry.delete(0, tk.END)
    # Inserta el valor por defecto "/home/" en el campo home
    self.home_entry.insert(0, "/home/")
    # Pone el shell al valor por defecto "/bin/bash"
    self.shell_combobox.set("/bin/bash")
    # Limpia la selección de la lista de grupos
    self.group_listbox.selection_clear(0, tk.END)
    # Actualiza la barra de estado para indicar que los campos fueron limpiados
    self.update_status("Campos limpiados")
```

4.2.2-FUNCIONALIDADES DE GRUPOS

Empiezo cargando la ***lista de grupos*** del sistema ejecutando el comando getent group, pero para ello ***limpié la tabla previa*** y agregué cada grupo con su información al widget visual.

```
def load_group_list(self):
    """Carga la lista de grupos"""
    try:
        # Elimina todos los elementos actuales del árbol de grupos para refrescar la lista
        for item in self.group_tree.get_children():
            self.group_tree.delete(item)

        # Ejecuta el comando 'getent group' para obtener la lista de grupos del sistema
        result = subprocess.run(['getent', 'group'], capture_output=True, text=True, encoding='utf-8')
        if result.returncode == 0:
            groups = result.stdout.splitlines()
            # Procesa cada línea para extraer nombre, GID y miembros del grupo
            for group in groups:
                parts = group.split(':')
                if len(parts) >= 4:
                    groupname = parts[0]
                    gid = parts[2]
                    members = parts[3] if len(parts) > 3 else ""
                    # Inserta la información del grupo en el árbol visual
                    self.group_tree.insert("", "end", values=(groupname, gid, members))
            # Barra de estado con la cantidad de grupos cargados
            self.update_status(f"Grupos cargados: {len(groups)}")
            logging.info("Lista de grupos actualizada")
        else:
            #Error de si el comando falló
            messagebox.showerror("Error", "Error al obtener grupos")
            logging.error(f"Error getent group: {result.stderr}")

    except Exception as e:
        # Excepciones mostrando mensaje de error y registrándolo
        messagebox.showerror("Error", f"Excepción: {str(e)}")
        logging.error(f"Excepción load_group_list: {str(e)}")
```

Como segundo paso, capturé el evento de selección de un ***grupo en el árbol***, extraje su información y actualicé los campos del formulario con el nombre y el GID del grupo seleccionado.

```
def on_group_select(self, event):
    """Evento al seleccionar un grupo"""
    selected = self.group_tree.selection()  # Obtuve el grupo seleccionado en el árbol de grupos
    if selected:
        group_data = self.group_tree.item(selected[0])['values']  # Extrae los valores del grupo seleccionado
        self.group_entry.delete(0, tk.END)  # Limpia el campo del nombre del grupo en el formulario
        self.group_entry.insert(0, group_data[0])  # Añade el nombre del grupo obtenido
        self.gid_entry.delete(0, tk.END)  # Borra el campo del GID en el formulario
        self.gid_entry.insert(0, group_data[1])  # Inserta el GID correspondiente del grupo seleccionado
```

A continuación, apliqué las ***validaciones*** para que el campo del nombre de grupo no estuviera vacío y mostré una advertencia si faltaba algo.

```
def validate_group_fields(self):
    """Valida los campos del formulario de grupo"""
    groupname = self.group_entry.get().strip()  # Eliminé espacios del nombre del grupo ingresado
    if not groupname:
        messagebox.showwarning("Validación", "Nombre de grupo requerido")  #Aadvertencias de si estaba vacío
        return False  # Finalicé la validación con resultado negativo
    return True  # Validación exitosa si el nombre no estaba vacío
```

A la hora de crear los grupos, primero procedí a validar los campos, verificar si el grupo ya existía y, si no, ***ejecutar groupadd*** para crearlo.

```
def create_group(self):
    """Crea un nuevo grupo"""
    if not self.validate_group_fields():  # Validé los campos del formulario
        return

    groupname = self.group_entry.get().strip()  # Nombre del grupo
    gid = self.gid_entry.get().strip()          # GID si se proporcionó

    try:
        if self.group_exists(groupname):  # Verifiqué si el grupo ya existía
            messagebox.showerror("Error", f"Grupo '{groupname}' ya existe")
            return

        cmd = ['sudo', 'groupadd']        # Comando para crear grupo
        if gid:
            cmd.extend(['-g', gid])
        cmd.append(groupname)             # Añade el nombre del grupo

        result = subprocess.run(cmd, capture_output=True, text=True, encoding='utf-8')  # Ejecuté el comando

        if result.returncode == 0:
            messagebox.showinfo("Éxito", f"Grupo '{groupname}' creado")  # Mostré mensaje de éxito
            logging.info(f"Grupo creado: {groupname}")
            self.load_group_list()        # Recargué la lista de grupos
            self.clear_group_fields()     # Limpié el formulario
        else:
            messagebox.showerror("Error", f"Error al crear grupo: {result.stderr}")  # Muestra si hay error
            logging.error(f"Error create_group: {result.stderr}")

    except Exception as e:
        messagebox.showerror("Error", f"Error: {str(e)}")  # Captura excepciones
        logging.error(f"Excepción create_group: {str(e)}")
```

Para la eliminación del grupo, verifico que se haya ingresado un nombre de grupo, compruebo si existía, pide confirmación y luego ejecuta ***groupdel***. Si se eliminó correctamente, muestra un mensaje, actualiza la lista de grupos y limpia los campos o en caso contrario, informa del error.

```
def delete_group(self):
    """Elimina un grupo"""
    groupname = self.group_entry.get().strip()  # Obtiene el nombre del grupo desde el campo de entrada
    if not groupname:
        messagebox.showwarning("Validación", "Seleccione un grupo")  # Comprueba si el campo estaba vacío
        return
    if not self.group_exists(groupname):  # Verifica si el grupo existe en el sistema
        messagebox.showerror("Error", "El grupo no existe")
        return
    if messagebox.askyesno(
        "Confirmar eliminación",
        f"¿Eliminar grupo '{groupname}'?",
        icon="warning"
    ):  # Confirmación antes de eliminar
        try:
            result = subprocess.run(
                ['sudo', 'groupdel', groupname],  # Ejecuta el comando para eliminar el grupo
                capture_output=True,
                text=True,
                encoding='utf-8'
            )
            if result.returncode == 0:
                messagebox.showinfo("Éxito", f"Grupo '{groupname}' eliminado")  # Mensaje de éxito
                logging.info(f"Grupo eliminado: {groupname}")
                self.load_group_list()      # Recarga la lista de grupos
                self.clear_group_fields()
            else:
                messagebox.showerror("Error", f"Error: {result.stderr}") #El mensaje de error
                logging.error(f"Error delete_group: {result.stderr}")
        except Exception as e:
            messagebox.showerror("Error", f"Error: {str(e)}")
            logging.error(f"Excepción delete_group: {str(e)}")
```

Ahora toca ponerse en la casuística de que el grupo exista, así que utilicé ***getent group*** para consultar si el grupo existía en el sistema. Si el comando se ejecutó correctamente, devuelve True, de lo contrario, devuelve False.

```
def group_exists(self, groupname):
    """Verifica si un grupo existe"""
    try:
        # Ejecuta el comando 'getent group <nombre_grupo>' para consultar si el grupo existe
        subprocess.run(
            ['getent', 'group', groupname],
            check=True,                     # Lanza excepción si el grupo no se encuentra
            stdout=subprocess.PIPE,
            stderr=subprocess.PIPE,
            encoding='utf-8'
        )
        return True  # Si el comando tuvo éxito, el grupo existe
    except subprocess.CalledProcessError:
        return False  # Si falló, el grupo no existe
```

Para terminar con los grupos, respecto al botón de **eliminar campos**, pongo estas líneas para limpiar todos los campos y que se actualice.

```
def clear_group_fields(self):
    """Limpia los campos de grupos"""
    self.group_entry.delete(0, tk.END)     # Elimina el contenido del campo de nombre de grupo
    self.gid_entry.delete(0, tk.END)       # Elimina el contenido del campo GID
    self.update_status("Campos de grupo limpiados")  # Actualiza la barra de estado
```

4.2.3-FUNCIONALIDADES DE LOGS

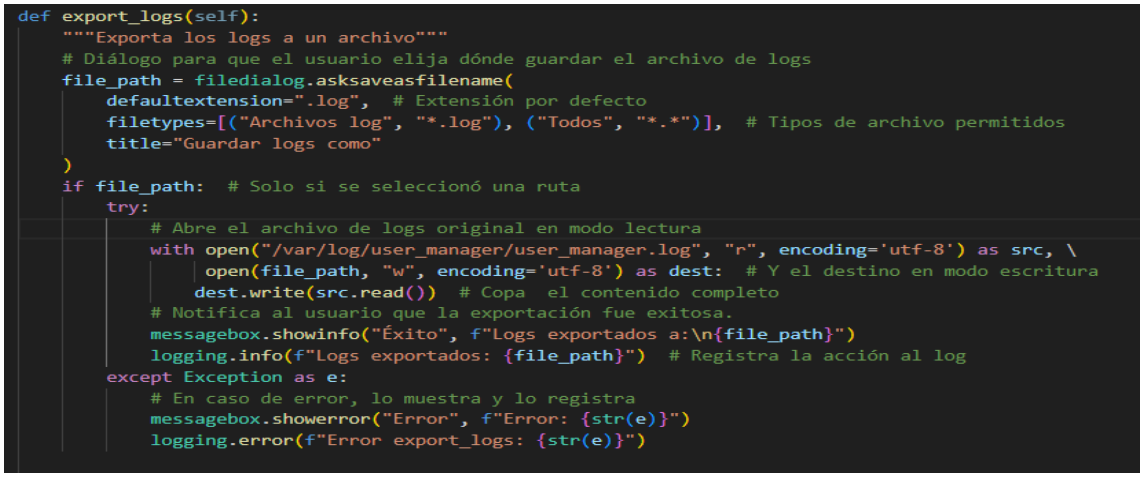
Comenzando con la parte de logs, como primer paso abrí ***el archivo de logs***, mostré su contenido en el widget de texto, actualicé la barra de estado y gestioné errores como archivo no encontrado o excepciones imprevistas.

```
def update_logs(self):
    """Actualiza los logs en la interfaz"""
    try:
        # Abre el archivo de logs y lee su contenido
        with open("/var/log/user_manager/user_manager.log", "r", encoding='utf-8') as f:
            log_content = f.read()
        # Habilita el widget de texto para editarlo
        self.log_text.config(state="normal")
        self.log_text.delete(1.0, tk.END)              # Limpia el contenido anterior
        self.log_text.insert(tk.END, log_content)      # Inserta el nuevo contenido del log
        self.log_text.config(state="disabled")         # Lo deshabilité para evitar edición
        self.log_text.yview(tk.END)                    # Desplaza la vista al final del texto
        self.update_status("Logs actualizados")        # Actualiza la barra de estado
    except FileNotFoundError:
        # Mensaje si el archivo no existe
        self.log_text.config(state="normal")
        self.log_text.delete(1.0, tk.END)
        self.log_text.insert(tk.END, "Archivo de logs no encontrado")
        self.log_text.config(state="disabled")
    except Exception as e:
        # Mensaje sobre si ocurrió un error inesperado
        self.log_text.insert(tk.END, f"Error: {str(e)}")
```

Para ***limpiar los logs***, verifica si el usuario querría ***borrar los logs***, en caso de que si, vacía el archivo correspondiente, le enseña un mensaje de aviso y registra la acción (si ocurrió un error, lo informa y lo registra también).

```
def clear_logs(self):
    """Limpia los logs"""
    # Confirma con el usuario si desea borrar los logs
    if messagebox.askyesno(
        "Confirmar",
        "¿Borrar todos los logs? Esta acción es irreversible",
        icon="warning"
    ):
        try:
            # Abre el archivo de logs en modo escritura para vaciarlo
            with open("/var/log/user_manager/user_manager.log", "w", encoding='utf-8') as f:
                f.write("")  # Lo deja en blanco
            self.update_logs()  # Actualiza la vista del log en la interfaz
            messagebox.showinfo("Éxito", "Logs borrados")  # Confirmación al usuario
            logging.info("Logs borrados manualmente")      # Registra la acción en el log del sistema
        except Exception as e:
            # En caso de error, enseña un mensaje y lo registra
            messagebox.showerror("Error", f"Error: {str(e)}")
            logging.error(f"Error clear_logs: {str(e)}")
```

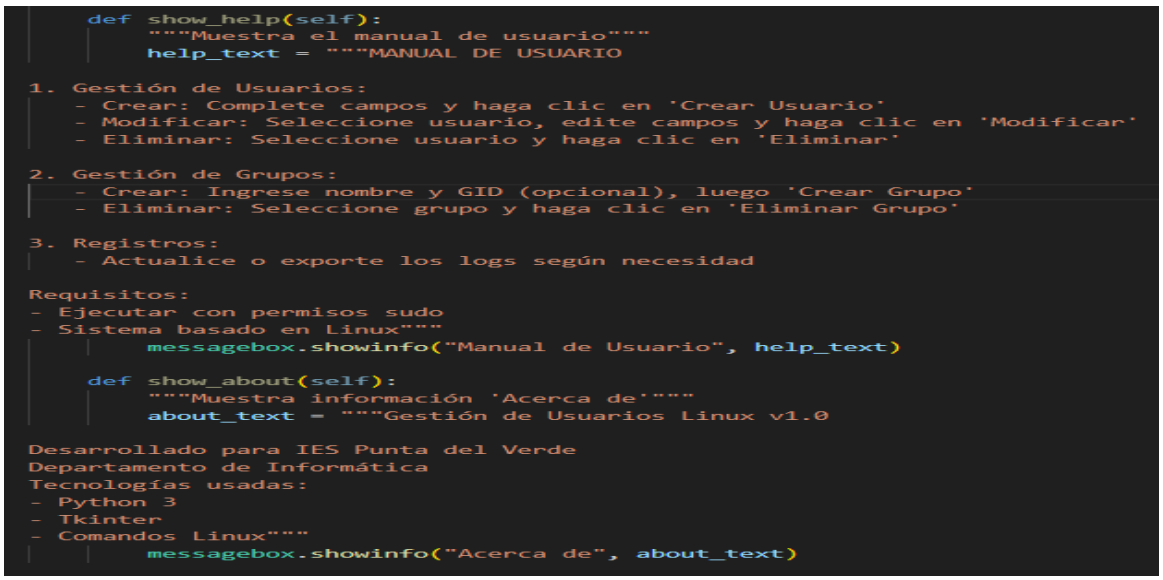
También añadí una función de exportar los logs lo cual le permite al usuario seleccionar una ubicación para ***exportar los logs***, copia el contenido del archivo de log original al destino elegido, notifica el éxito o reporta cualquier error que surgiera durante el proceso.



```
def export_logs(self):
    """Exporta los logs a un archivo"""
    # Diálogo para que el usuario elija dónde guardar el archivo de logs
    file_path = filedialog.asksaveasfilename(
        defaultextension=".log",  # Extensión por defecto
        filetypes=[("Archivos log", "*.log"), ("Todos", "*.*")],  # Tipos de archivo permitidos
        title="Guardar logs como"
    )
    if file_path:  # Solo si se seleccionó una ruta
        try:
            # Abre el archivo de logs original en modo lectura
            with open("/var/log/user_manager/user_manager.log", "r", encoding='utf-8') as src, \
                 open(file_path, "w", encoding='utf-8') as dest:  # Y el destino en modo escritura
                dest.write(src.read())  # Copia  el contenido completo
            # Notifica al usuario que la exportación fue exitosa.
            messagebox.showinfo("Éxito", f"Logs exportados a:\n{file_path}")
            logging.info(f"Logs exportados: {file_path}")  # Registra la acción al log
        except Exception as e:
            # En caso de error, lo muestra y lo registra
            messagebox.showerror("Error", f"Error: {str(e)}")
            logging.error(f"Error export_logs: {str(e)}")
```

4.2.4-FUNCIONALIDADES DEL MENÚ

Por último, respecto a las funcionalidades, hice dos botones para mostrar el **Manual de Usuario** donde creé un texto con las instrucciones principales de uso del sistema que lo muestra mediante un cuadro de diálogo informativo para que el usuario y pueda consultar el manual de manera rápida dentro de la app. También hice un ***Acerca de*** que contiene las tecnologías utilizadas en la aplicación.



```
    def show_help(self):
        """Muestra el manual de usuario"""
        help_text = """MANUAL DE USUARIO

1. Gestión de Usuarios:
   - Crear: Complete campos y haga clic en 'Crear Usuario'
   - Modificar: Seleccione usuario, edite campos y haga clic en 'Modificar'
   - Eliminar: Seleccione usuario y haga clic en 'Eliminar'

2. Gestión de Grupos:
   - Crear: Ingrese nombre y GID (opcional), luego 'Crear Grupo'
   - Eliminar: Seleccione grupo y haga clic en 'Eliminar Grupo'

3. Registros:
   - Actualice o exporte los logs según necesidad

Requisitos:
- Ejecutar con permisos sudo
- Sistema basado en Linux"""
        messagebox.showinfo("Manual de Usuario", help_text)

    def show_about(self):
        """Muestra información 'Acerca de'"""
        about_text = """Gestión de Usuarios Linux v1.0

Desarrollado para IES Punta del Verde
Departamento de Informática
Tecnologías usadas:
- Python 3
- Tkinter
- Comandos Linux"""
        messagebox.showinfo("Acerca de", about_text)
```

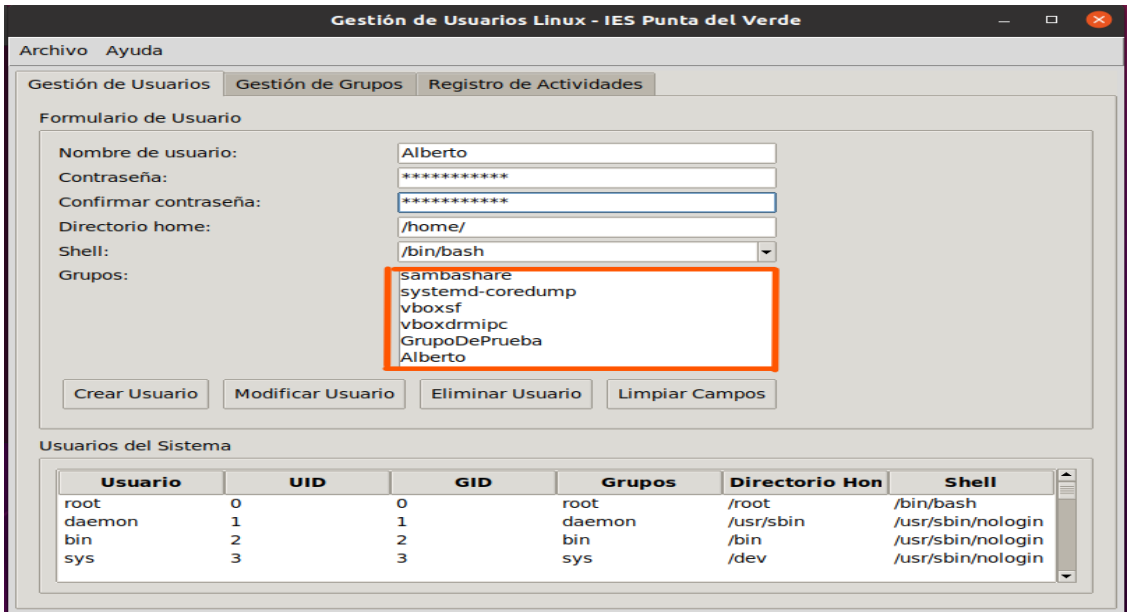
CAPÍTULO 5: PRUEBAS DE EJECUCIÓN

5.1.-CASOS DE PRUEBA

5.2.-CASOS DE PRUEBA (USUARIOS)

A la hora de ***crear el usuario*** que queramos, podemos elegir directamente que shell le queremos poner con un despliegue de opciones para elegir el shell adecuado.

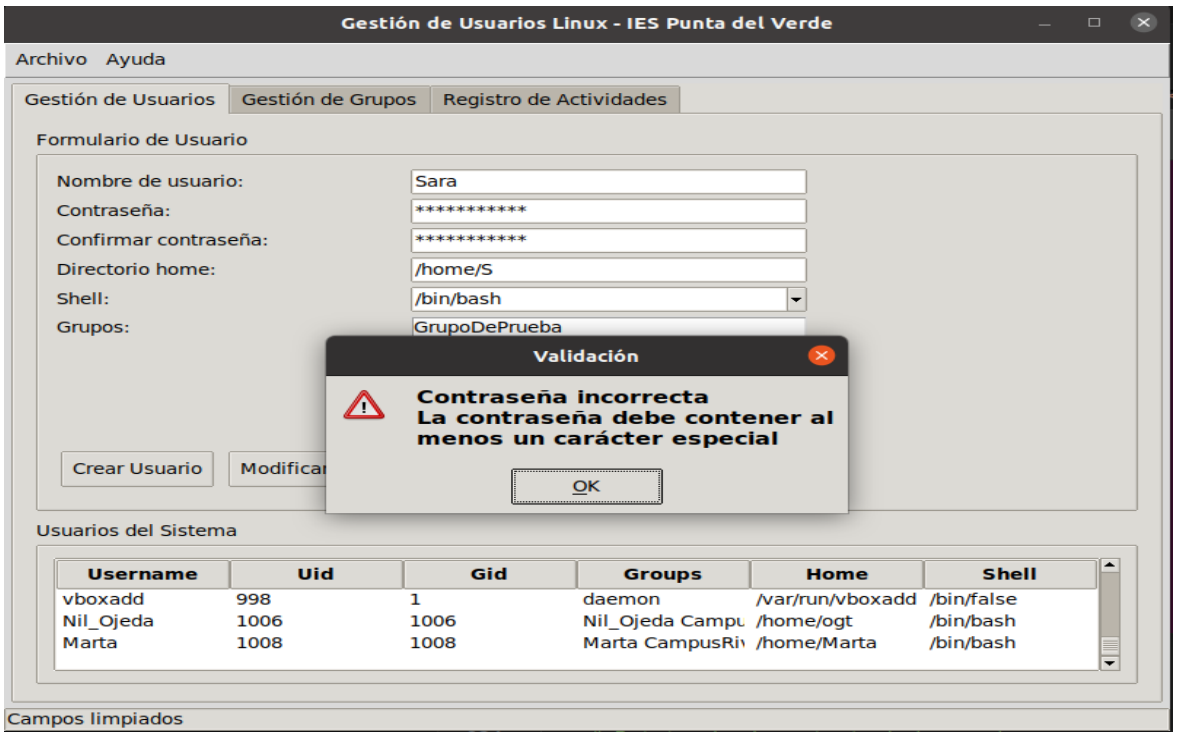
Además, podremos marcar los grupos a los que queramos (incluso los que nosotros creemos con la herramienta de ***Gestión de Grupos***) que el usuario esté añadido:



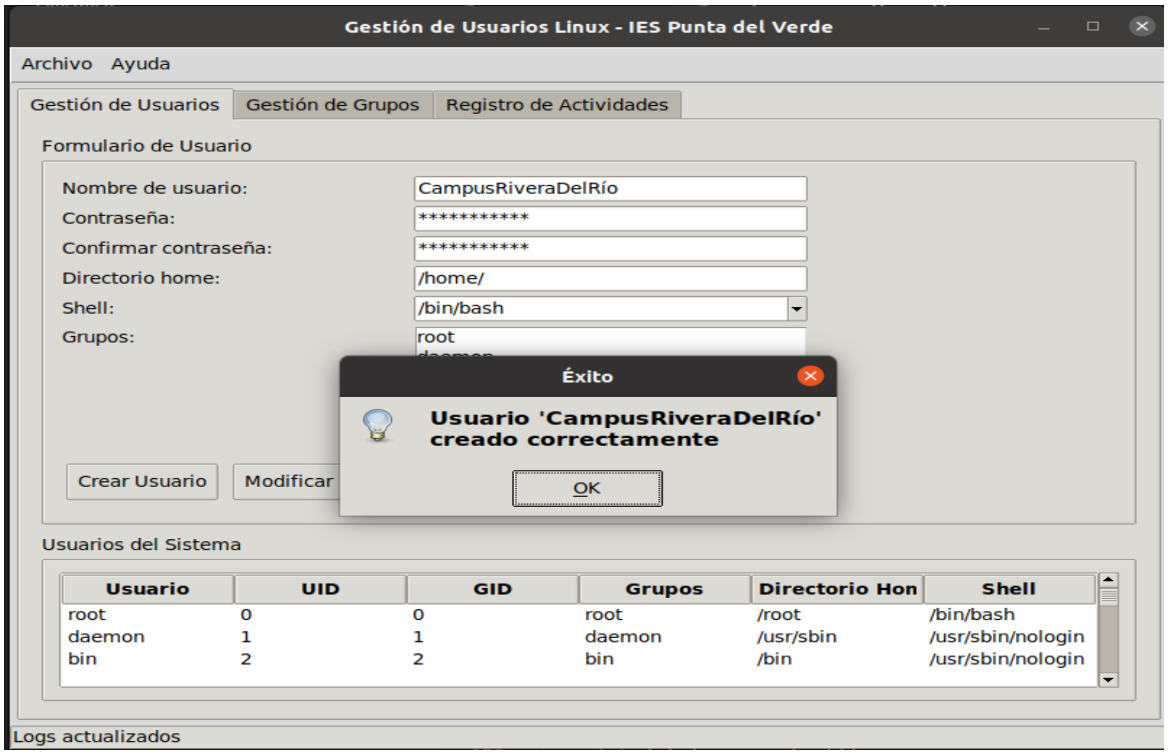
Al poner una contraseña que no cumpla el valor de la validación establecida, nos saltará el siguiente mensaje:

Si las contraseñas en su confirmación no coinciden, tampoco nos dejará y nos saldrá de nuevo otro cuadro de mensaje de ***error de validación***:

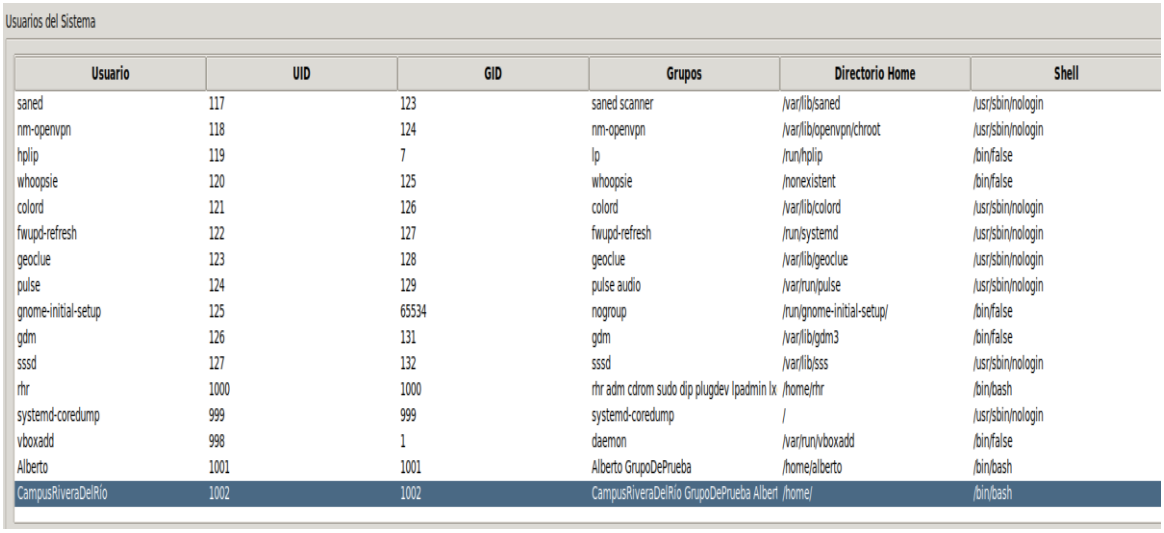
La contraseña debe tener una minúscula, una mayúscula y un carácter especial además de los 8 caracteres mínimos.



Una vez cumplimos todas estas validaciones y al darle al botón de "***Crear Usuario***" nos saltará este recuadro donde nos pone que el usuario se ha creado correctamente:

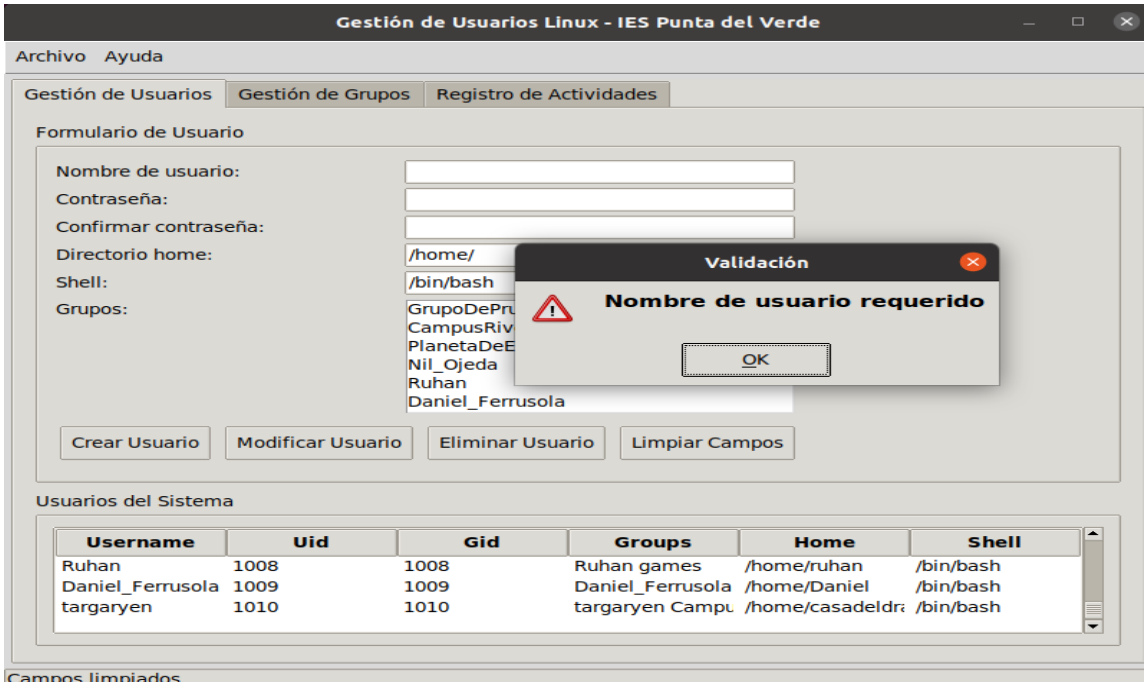


Una vez creado, en el apartado que hemos podido ver todo este tiempo en la parte inferior del menú, llamado ***Usuarios del sistema***, observamos que actualmente nos aparece el usuario que acabamos de crear.

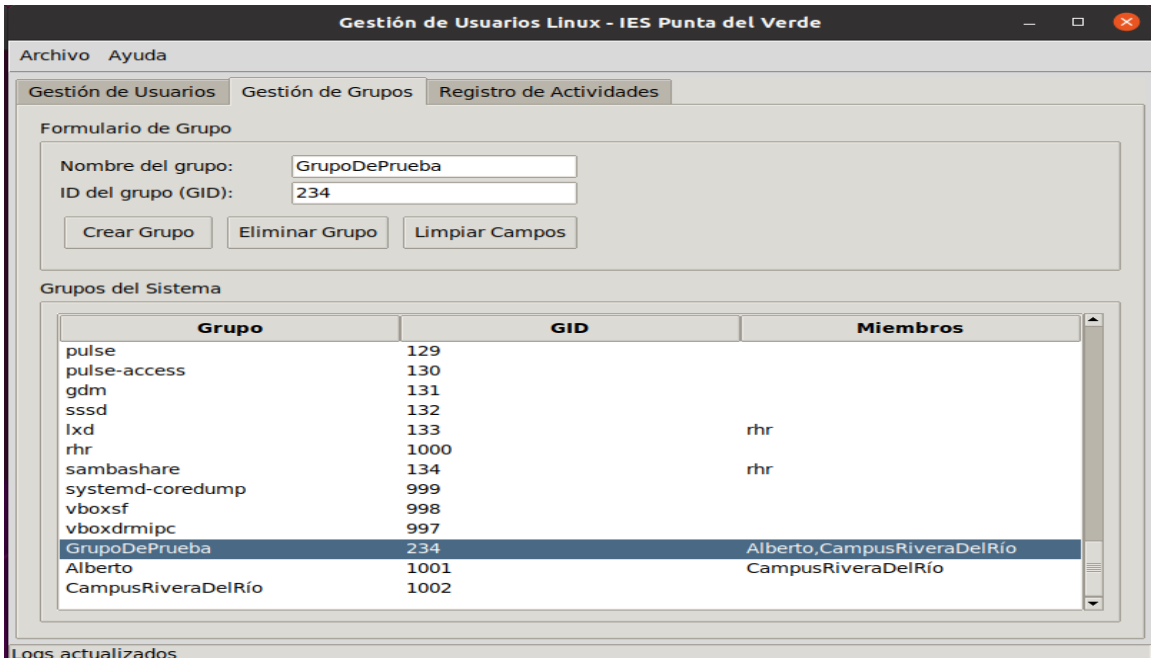

Usuarios del Sistema

Usuario	UID	GID	Grupos	Directorio Home	Shell
saned	117	123	saned scanner	/var/lib/saned	/usr/sbin/nologin
nm-openvpn	118	124	nm-openvpn	/var/lib/openvpn/chroot	/usr/sbin/nologin
hplip	119	7	lp	/run/hplip	/bin/false
whoopsie	120	125	whoopsie	/nonexistent	/bin/false
colord	121	126	colord	/var/lib/colord	/usr/sbin/nologin
fwupd-refresh	122	127	fwupd-refresh	/run/systemd	/usr/sbin/nologin
geoclue	123	128	geoclue	/var/lib/geoclue	/usr/sbin/nologin
pulse	124	129	pulse audio	/var/run/pulse	/usr/sbin/nologin
gnome-initial-setup	125	65534	nogroup	/run/gnome-initial-setup/	/bin/false
gdm	126	131	gdm	/var/lib/gdm3	/bin/false
sssd	127	132	sssd	/var/lib/sss	/usr/sbin/nologin
rhr	1000	1000	rhr adm cdrom sudo dip plugdev lpadmin lx	/home/rhr	/bin/bash
systemd-coredump	999	999	systemd-coredump	/	/usr/sbin/nologin
vboxadd	998	1	daemon	/var/run/vboxadd	/bin/false
Alberto	1001	1001	Alberto GrupoDePrueba	/home/alberto	/bin/bash
CampusRiveraDelRio	1002	1002	CampusRiveraDelRio GrupoDePrueba Albert	/home/	/bin/bash

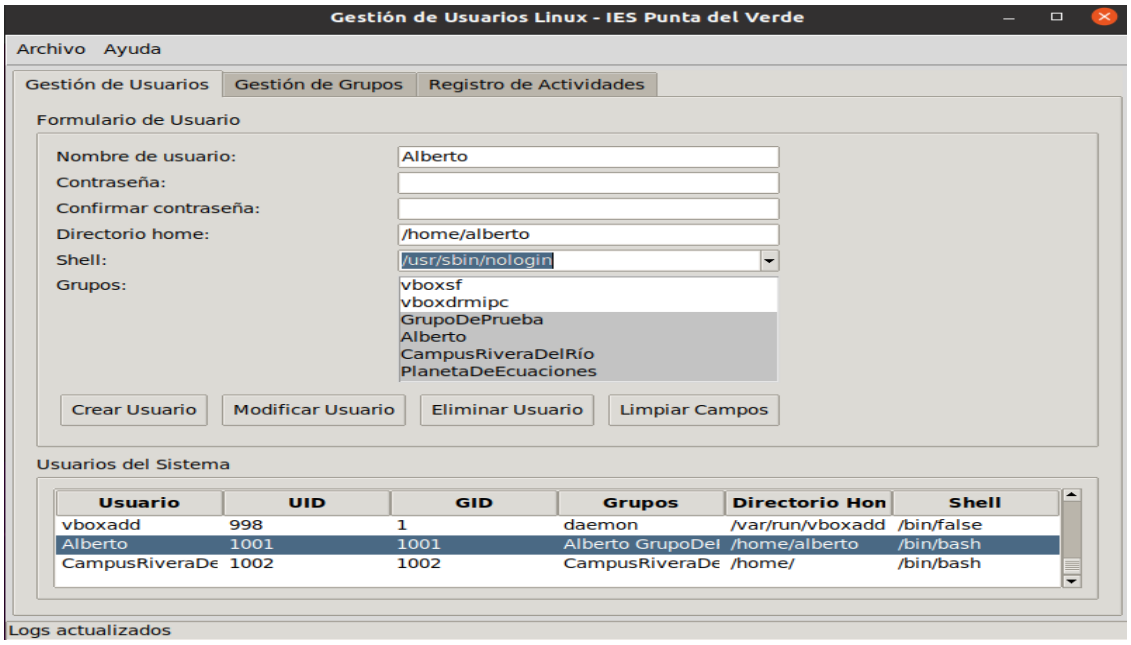
En caso de crear un usuario sin nombre también nos aparecerá:



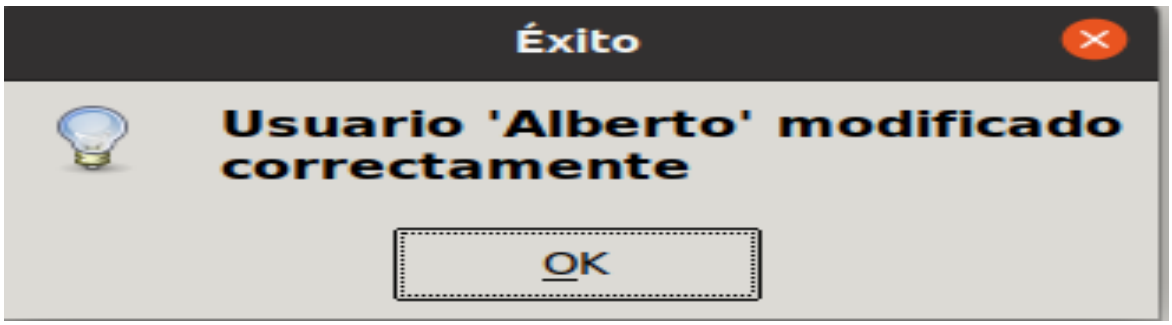
Dicha información respecto a los grupos, los cuales, es participe el usuario se muestra también en la pestaña en la parte superior de ***Gestión de Grupos.***



En cuanto a ***Modificar un Usuario***, sería tan simple como seleccionar al usuario que queremos modificar en la Lista de Usuarios en la parte inferior del menú y cambiar los atributos que necesitemos. Aunque ***no cambiemos las contraseñas***, el usuario seguirá estando protegido por la primera que se le asignó.



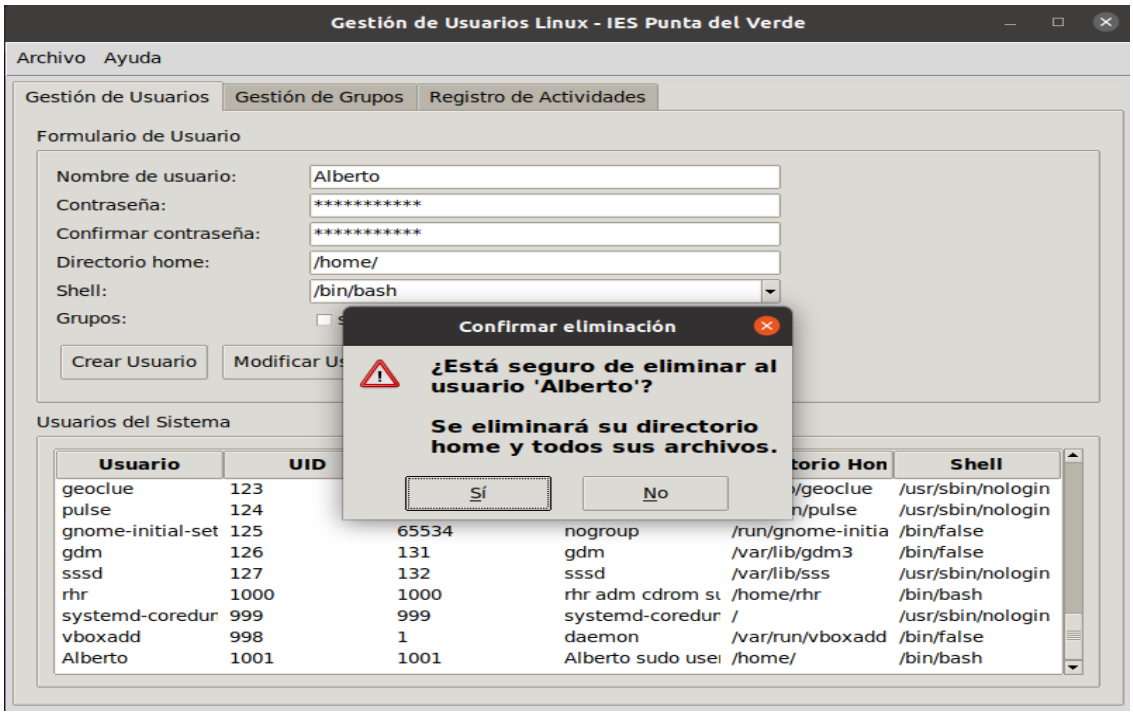
Una vez aplicados los cambios, nuevamente nos saldrá otro aviso de que se ha llevado a cabo exitosamente.



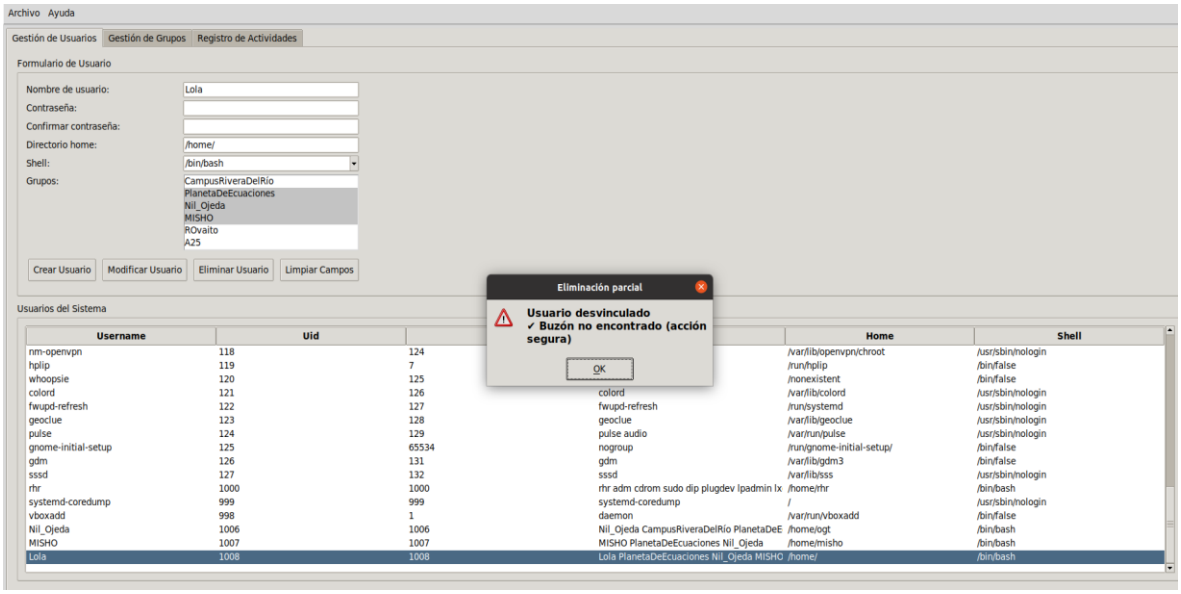
De forma y manera que en ***La lista de Grupos*** de la sección Gestión de Grupos, los cambios se habrían aplicado.

GrupoDePrueba	234	Alberto
Alberto	1001	Alberto
CampusRiveraDelRío	1002	Alberto
PlanetaDeEcuaciones	777360	CampusRiveraDelRío,Alberto

Respecto al ***botón de eliminar*** a dicho usuario, nos aparecerá el mensaje de si estamos seguros de eliminar a dicho usuario y que se eliminarán todos los directorios y archivos relacionado con el usuario.

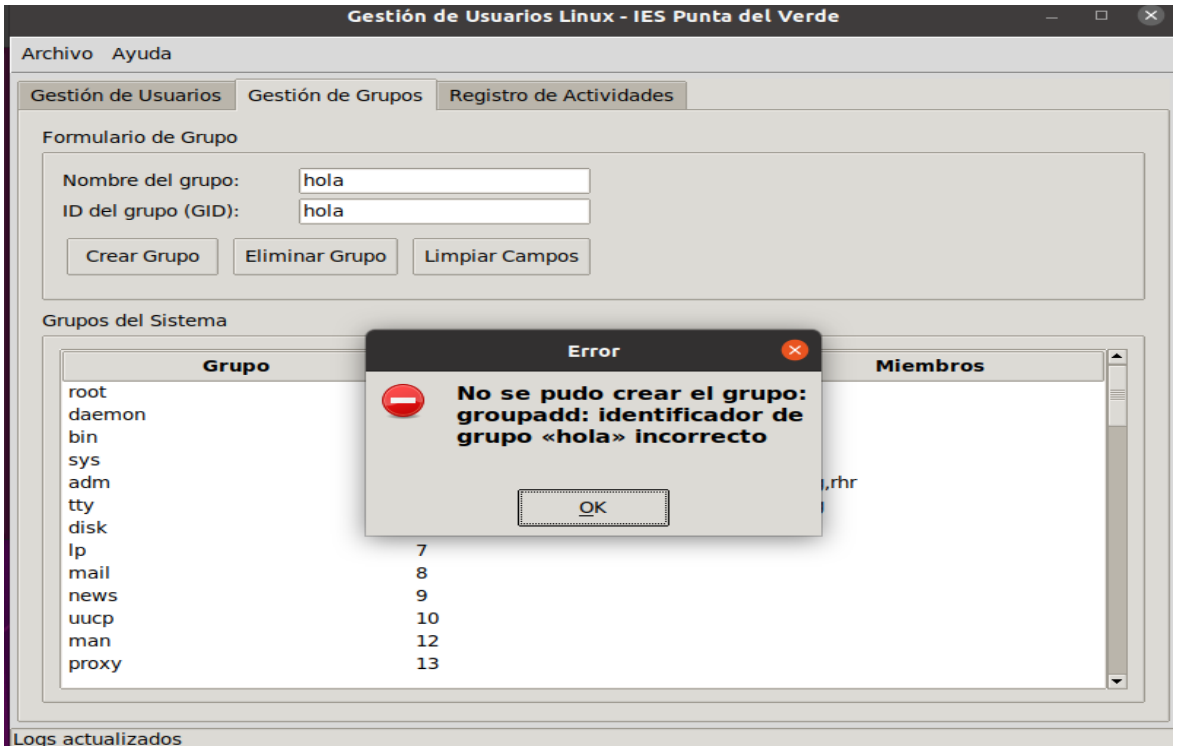


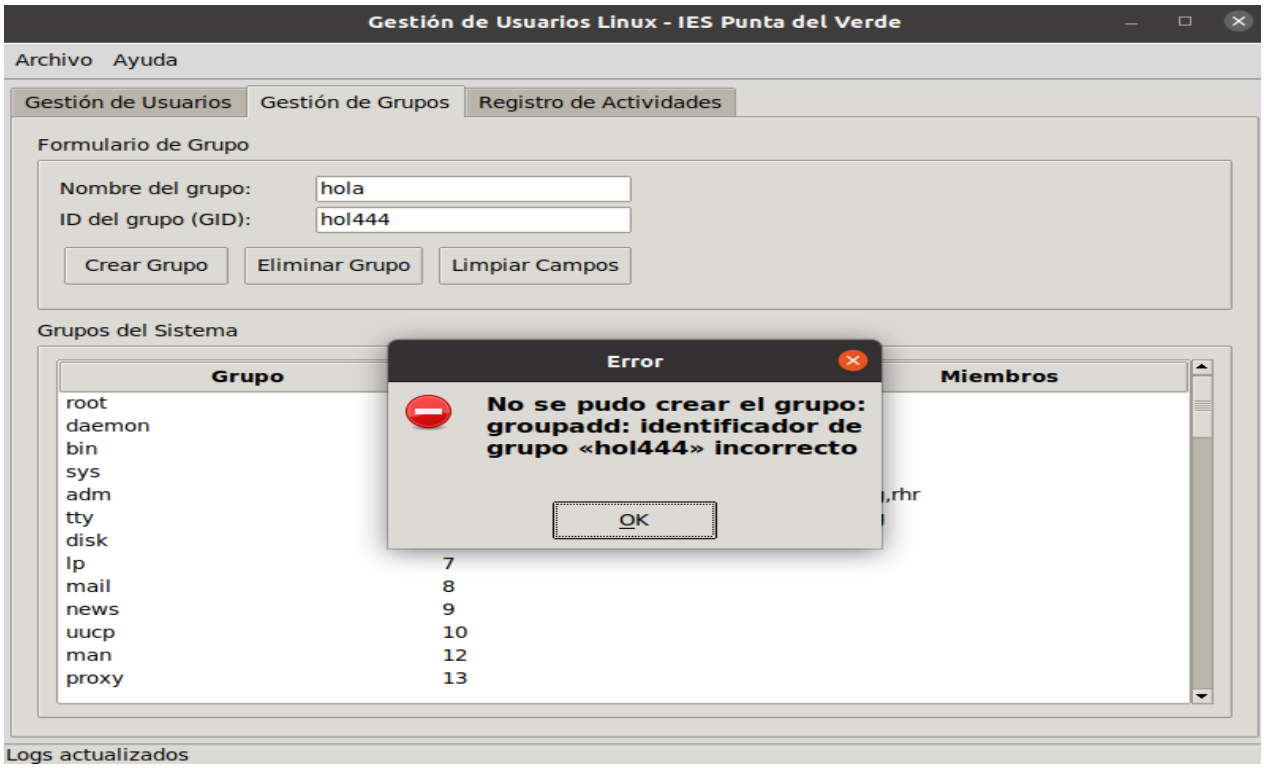
Cuando el/la usuario/a ***no dispone de directorio personal*** y se crea con el por defecto, nos saldrá este mensaje al eliminarlos:



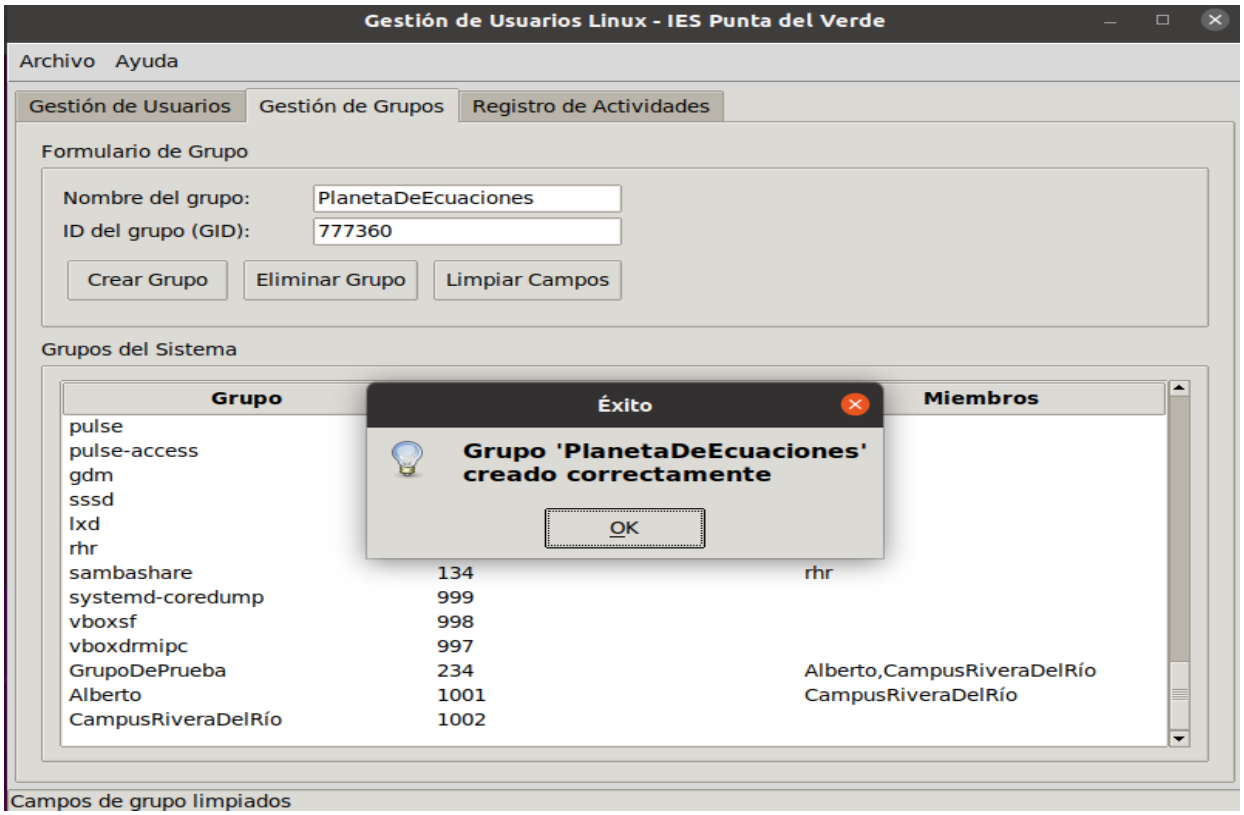
5.3.-CASOS DE PRUEBA (GRUPOS)

Para empezar con este apartado, si ponemos en el campo de ***Id del Grupo*** algún valor que no sea en su totalidad numérico, nos saltará un error de validación que nos dirá que los campos están mal establecidos.

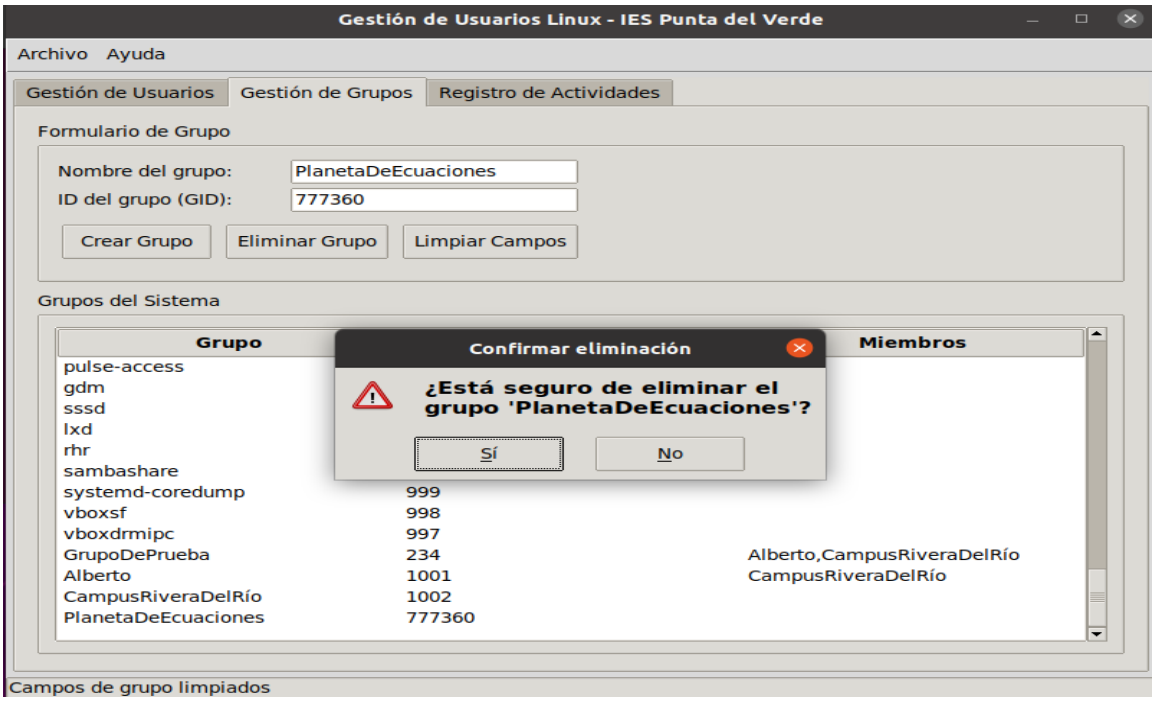




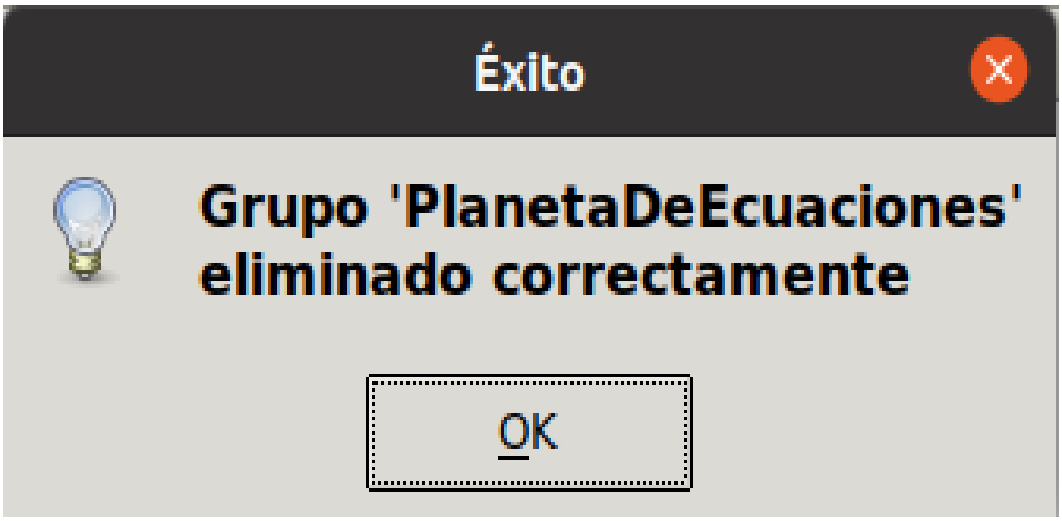
En caso contrario, nos encontraremos con un mensaje de éxito en la operación que acabamos de realizar, añadiendo dicho grupo a la lista de ***Grupos del Sistema*** en la parte inferior.



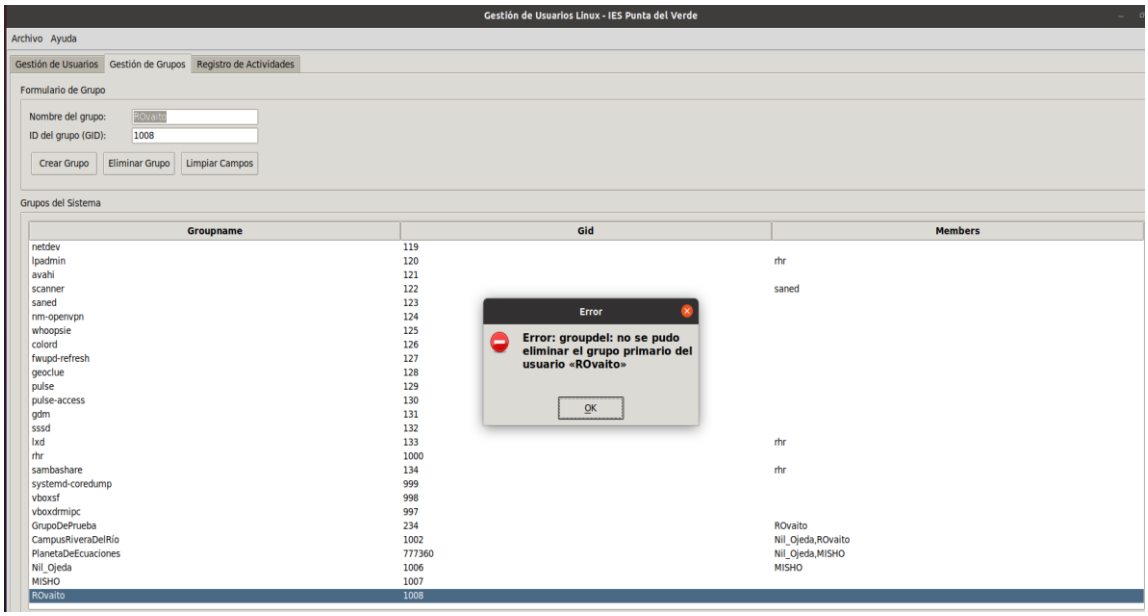
A la hora de disponernos a ***eliminar un grupo***, nos aparecerá otro cuadro de alerta por si estamos verdaderamente seguros de lo que vamos a eliminar.



Si le damos a la opción de Sí, desaparecerá de la lista de Grupos del Sistema y nos encontraremos con esta otra alerta:

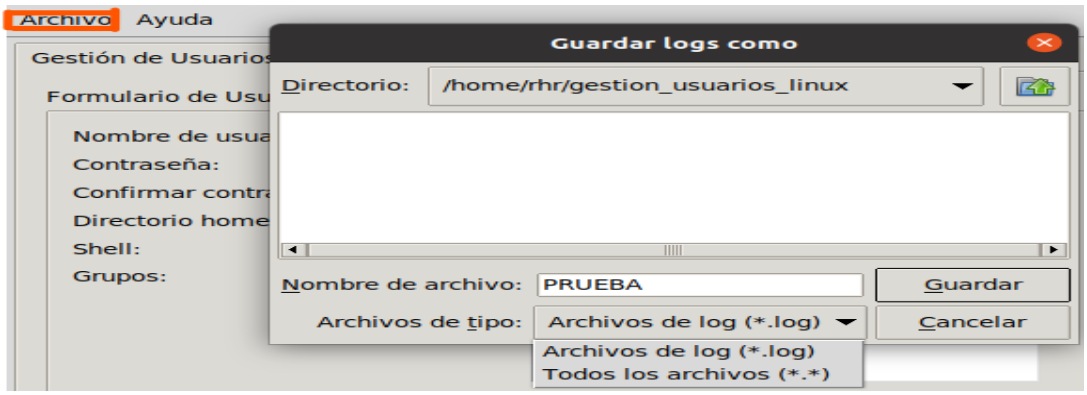


Asimismo, no podremos borrar el ***grupo principal de un usuario existente***:

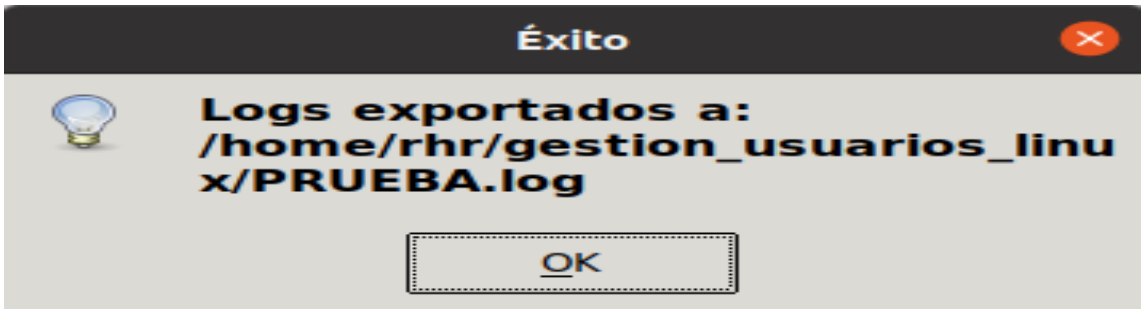


5.4.-CASOS DE PRUEBA (LOGS)

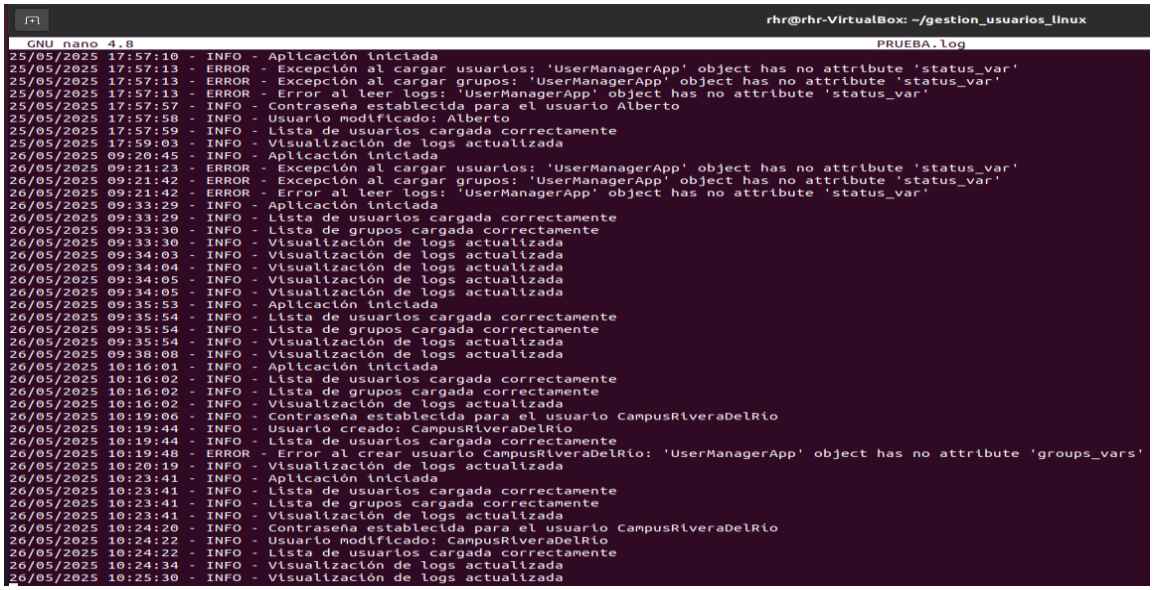
En el menú en la parte superior izquierda encontramos un apartado de Archivo donde podremos exportar los logs con los que hayamos trabajado hasta el momento.



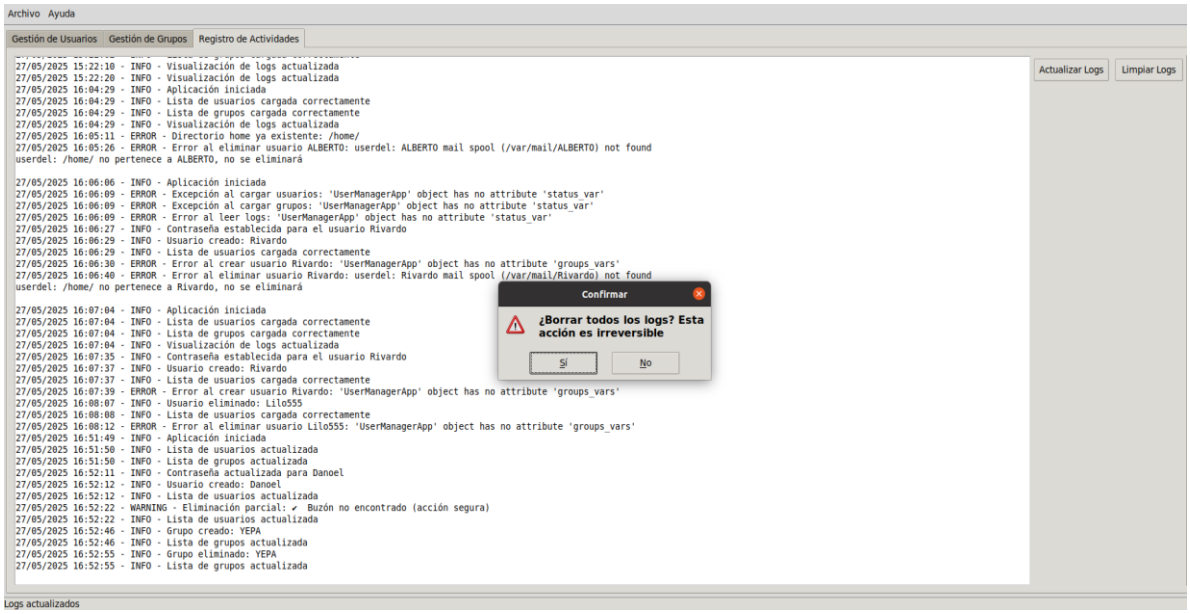
En caso correcto de haberse guardado correctamente nos saldrá este aviso, indicándonos donde se ha descargado:



Si nos dirigimos a ver el contenido del archivo veremos que todo está correctamente.



Si optamos por la opción de ***eliminar el registro de logs***, nos saldrá algo como esto, indicándonos que no vamos a poder recuperar dicha información.



Y por último este recuadro:

CAPÍTULO 6: CONCLUSIONES

6.1.- CONCLUSIONES

El objetivo de este proyecto ha sido crear una herramienta completa y funcional para la ***gestión de usuarios en sistemas Linux***, diseñada para situaciones en las que se necesita un control efectivo sin tener que utilizar únicamente el terminal.
He sido capaz de hacer una ***interfaz gráfica sencilla*** y fácil de usar con Python y Tkinter que permite añadir, cambiar y eliminar usuarios, así como gestionar sus grupos y permisos.

Entre los éxitos que he logrado están:

- ✓ La unificación de ***comandos del sistema con subproceso***, manteniendo la compatibilidad con funciones de ***Linux estándar***.

- ✓ La representación estructurada y clara de usuarios y grupos como ***componentes gráficos***.

- ✓ El registro detallado de ***todas las operaciones realizadas***, en forma de auditoría.

- ✓ La configuración de la aplicación para utilizarse por ***técnicos inexpertos*** en la línea de comandos.

✓ Este proyecto ha demostrado que es posible ***modernizar la gestión de sistemas Linux*** tradicionales mediante soluciones accesibles y visuales sin sacrificar ni potencia ni seguridad.

6.2.- PROPUESTAS FUTURAS

Aunque la aplicación actual es estable y funciona bien, hay varias áreas de mejora y desarrollo que pueden estudiarse para versiones posteriores:

- ***Internacionalización:*** Brindar soporte para múltiples idiomas para que la interfaz se ajuste automáticamente al idioma del sistema o incluir una opción en la que se dé al usuario la opción de seleccionar un idioma apropiado en un menú.

- ***Gestión remota:*** Añadir la funcionalidad de gestionar usuarios en servidores remotos vía SSH para proporcionar una gestión centralizada de múltiples máquinas.

- ***Compatibilidad con LDAP:*** Aportar integración con servicios de directorio como LDAP o Active Directory para hacer la utilidad más útil en redes académicas o corporativas.

- ***Exportación/Importación de Configuraciones:*** Ofrecer opciones para exportar la configuración actual de grupos y usuarios a archivos. json o .csv, para facilitar la replicación de un entorno o la restauración instantánea.

- ***Notificaciones del sistema:*** Incluir un mecanismo de notificación (por ejemplo, duplicación, creación, usuario que alcanza la cuota), que proporcione información inmediata.

- ***Detalle del registro de auditoría de seguridad por técnico:*** Ampliar el sistema de registro actual para rastrear qué técnico realizó cada acción, incluyendo marcas de tiempo y, opcionalmente, firmas digitales para auditoría.
- ***Empaquetador e instalador:*** Empaquetar la aplicación como un paquete .deb instalable para distribuir e instalar la aplicación en sistemas basados en Debian/Ubuntu.

CAPÍTULO 7: BIBLIOGRAFÍA Y REFERENCIAS

7.1.-REFERENCIAS BIBLIOGRÁFICAS.

- **Documentación oficial de Python:** https://docs.python.org/3/
- **Subprocesos:** https://docs.python.org/3/library/subprocess.html
- **Tkinter**:https://docs.python.org/3/library/tkinter.html, https://realpython.com/python-gui-tkinter/
- **Man de Linux:** https://man7.org/linux/man-pages/
- **PEP 8 para el estilo:** https://peps.python.org/pep-0008/
- **File handling:** https://stackoverflow.com/questions

ANEXO 1: MANUAL DE INSTALACIÓN

La aplicación requiere un ***sistema operativo Linux*** y tener instalado Python 3.8 o superior, además de contar con ***privilegios de administrador*** (root o sudo) para ejecutar correctamente las operaciones. Aunque no es obligatorio, yo recomiendo disponer de conexión a internet en caso de necesitar ***instalar dependencias adicionales*** respecto a la primera vez.

Mi recomendación como primer paso sería actualizar el sistema.

```
rhr@rhr-VirtualBox:~$ sudo apt update && sudo apt upgrade
```

Una vez hecho esto, si no lo tenemos anteriormente instalado en el equipo tenemos que instalar ***Python y Tkinter***.

```
rhr@rhr-VirtualBox:~$ sudo apt install python3 python3-tk
```

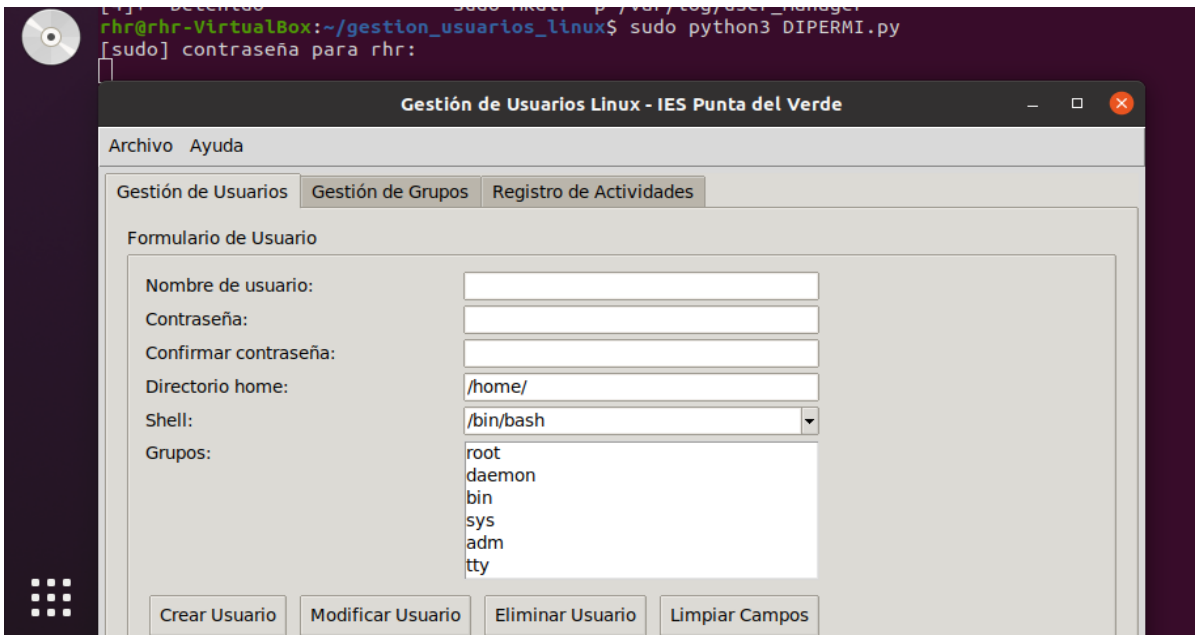
De forma consecutiva, ***creamos el directorio*** donde queramos utilizar la app. En dicho directorio colocamos el archivo y le otorgamos permisos de ejecución.

```
rhr@rhr-VirtualBox:~$ mkdir gestion_usuarios_linux
rhr@rhr-VirtualBox:~$ cd gestion_usuarios_linux
rhr@rhr-VirtualBox:~/gestion_usuarios_linux$ touch user_manager.py
rhr@rhr-VirtualBox:~/gestion_usuarios_linux$ chmod +x user_manager.py
rhr@rhr-VirtualBox:~/gestion_usuarios_linux$
```

Es posible que la carpeta de los logs no exista así que también podemos probar a crearla.

```
rhr@rhr-VirtualBox:~/gestion_usuarios_linux$ sudo mkdir -p /var/log/user_manager
rhr@rhr-VirtualBox:~/gestion_usuarios_linux$ sudo chmod 755 /var/log/user_manager
```

En mi caso, el archivo de mi aplicación se llama DIPERMI, el cual es el nombre con el que quiero bautizar mi aplicación. Lo importante aquí es que para ejecutar el programa usaremos ***sudo pyhon3 "nombre.py"*** y se abriría el programita. **IMPORTANTE** siempre usar **sudo**.



ANEXO 2: MANUAL DE USUARIO

Acerca de mi aplicación, he desarrollado ***tres pestañas principa***les que te voy a enseñar para que entiendas como funciona o qué presenta cada funcionalidad:

1-GESTIÓN DE USUARIOS

En esta pestaña encontrarás un formulario donde verás a primera vista esta serie de características:

- **Nombre de usuario:** Obligatorio.
- **Contraseña y confirmación:** Mínimo 8 caracteres, entre ellos una mayúscula, otra minúscula y un carácter especial. Si no pones contraseña no creará al usuario.
- **Directorio Home:** Por defecto creará el /home/”usuario”.
- **Shell:** seleccionable entre varias opciones.
- **Grupos:** se pueden seleccionar múltiples grupos existentes del sistema.

Además, justo debajo de todo lo que acabo de mencionar arriba, se encontrarán los ***botones*** que constan de:

- **Crear Usuario:** Agrega un nuevo usuario.
- **Modificar Usuario:** Cambia los datos de un usuario existente.
- **Eliminar Usuario:** Borra al usuario seleccionado.
- **Limpiar Campos:** Reinicia el formulario.

Justo debajo de todo esto encontraremos la ***tabla de usuarios*** la cual muestra los usuarios actuales con columnas para nombre, UID, GID, grupos, home y shell. Al hacer clic en uno, sus datos se cargan en el formulario.

2-GESTIÓN DE GRUPOS

La pestaña de Gestión de grupos permite crear y eliminar grupos del sistema, por ello verás dos campos: Nombre del Grupo que es obligatorio y GID el cual es un ID personal para añadir a dicho grupo.

Justo debajo tenemos los tres botones que funcionan igual que en la pestaña de usuarios, pero con grupos:

- **Crear Grupo:** Agrega un nuevo grupo.
- **Eliminar Grupo:** Borra al grupo seleccionado.
- **Limpiar Campos:** Reinicia el formulario.

3-REGISTRO DE ACTIVIDADES

En esta pestaña podremos visualizar el registro de auditoría de nuestro archivo log. Esta pestaña consta de tres botones:

- **Actualizar Logs:** Recarga el contenido.
- **Limpiar Logs:** Borra el contenido del archivo.

4- BOTONES DE ARCHIVO Y AYUDA

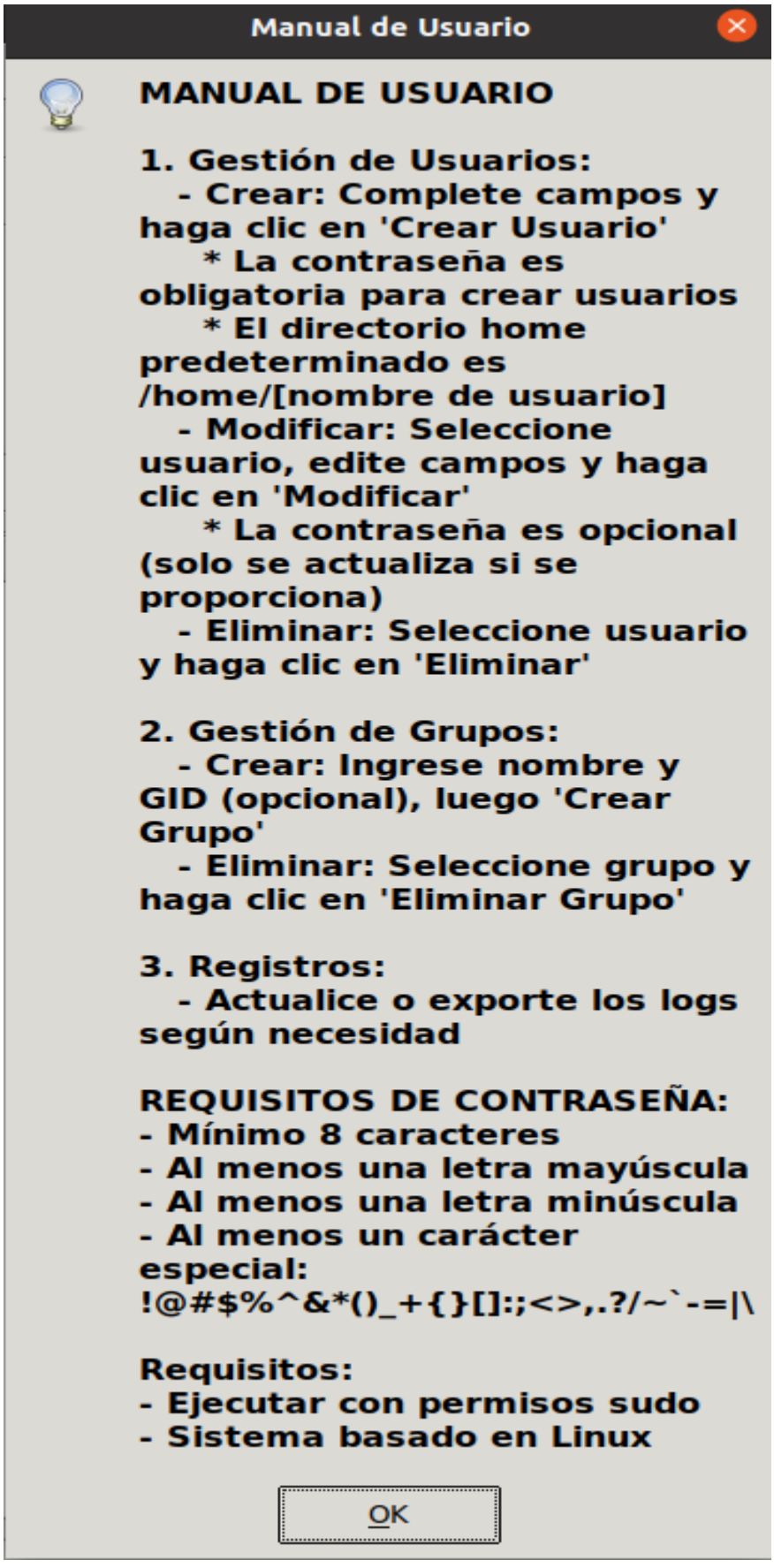
Si nos fijamos, en la parte superior izquierda podemos ver dos botones los cuales se desglosan; ***Archivo y Ayuda***.

Con botón de **"Archivo"** podemos contar para:

- **Exportar Logs:** Nos permite guardar una copia del log en otra ubicación.
- **SALIR:** Cierra el programa automáticamente.

Y sobre "Ayuda" decir que este será el botón que actúe como una guía rápida y simplificada dentro de la aplicación por si nos surge alguna duda. Se divide en:

- **Manual de Usuario:** Muestra un cuadro con la información más relevante de la app:


Manual de Usuario
MANUAL DE USUARIO
1. Gestión de Usuarios:
- Crear: Complete campos y haga clic en 'Crear Usuario'
* La contraseña es obligatoria para crear usuarios
* El directorio home predeterminado es /home/[nombre de usuario]
- Modificar: Seleccione usuario, edite campos y haga clic en 'Modificar'
* La contraseña es opcional (solo se actualiza si se proporciona)
- Eliminar: Seleccione usuario y haga clic en 'Eliminar'
2. Gestión de Grupos:
- Crear: Ingrese nombre y GID (opcional), luego 'Crear Grupo'
- Eliminar: Seleccione grupo y haga clic en 'Eliminar Grupo'
3. Registros:
- Actualice o exporte los logs según necesidad
REQUISITOS DE CONTRASEÑA:
- Mínimo 8 caracteres
- Al menos una letra mayúscula
- Al menos una letra minúscula
- Al menos un carácter especial:
!@#$%^&*()_+{}[]:;<>,.?/~`-=|\
Requisitos:
- Ejecutar con permisos sudo
- Sistema basado en Linux
OK

- **Acerca De:** Es simplemente un cuadro con las tecnologías usadas: