TAREA EVALUABLE T1 Ciberseguridad :

EJ 1

- Reemplacé la consulta por **prepared statements** para separar datos de la instrucción SQL.
- Uso de password_verify() para validar contraseñas contra hashes almacenados. Si el hash queda obsoleto, se re-hashea automáticamente.
- Mensajes genéricos de error para no revelar información sobre la existencia de cuentas.
- Límite simple de intentos y bloqueo temporal tras X intentos.
- Registro básico de intentos (archivo login_attempts.log).
- Regeneración de ID de sesión tras autenticación.

Ventaja principal
Las entradas del usuario ya no pueden alterar la estructura de la consulta SQL; con ello se elimina la posibilidad de inyección y se mejora la seguridad de las contraseñas.

Pruebas realizadas

- Login con credenciales válidas: OK.
- Login con contraseña incorrecta: OK (contador incrementado).
- Intento SQLi con ' OR '1'='1' --: no permite acceso.
- Bloqueo temporal tras varios intentos fallidos: OK.
- Registro de intentos creado y verificable.

EJ 2

Qué pasaba
El script usaba id de la URL directamente en la consulta. Aunque parezca número, alguien podía enviar texto con SQL y manipular la consulta (inyección).

Qué hice

- Validé id como entero (si no es número devuelve error).
- Usé *prepared statements* para que el valor no pueda alterar la consulta.
- Mensajes claros: 400 si el id es inválido; errores internos registrados.
- Mostré los datos con salida segura para evitar problemas en la página.

Pruebas rápidas

- Antes: ...?id=1 OR 1=1 podía devolver datos.
- Ahora: ...?id=1 OR 1=1 devuelve **400** y no ejecuta la inyección.
- ...?id=1 sigue mostrando el perfil cuando existe.

Recomendación corta
Siempre validar y usar consultas preparadas, aunque el dato parezca numérico. Añade logs y límites si el servicio es público.

EJ 3

- El problema era que el término buscado se imprimía en la página sin protección, así que si alguien enviaba código HTML/JS ese código se ejecutaba en el navegador de la víctima.
- La solución: usar htmlspecialchars() al mostrar el texto. Eso convierte caracteres especiales (<, >, " , ', &) en texto plano, de modo que el navegador los muestra en vez de ejecutar código.

Por qué ENT_QUOTES y UTF-8:

- ENT_QUOTES hace que tanto comillas dobles (") como simples (') se conviertan en entidades. Eso evita vectores XSS que usan comillas dentro de atributos HTML.
- Especificar 'UTF-8' asegura que la conversión respete la codificación correcta de la página; si no se pone y la codificación no coincide, puede abrirse una ventana para ataques raros o mostrar caracteres de forma incorrecta.

EJ 4

Problema detectado:
El código original guardaba comentarios sin protección y los mostraba sin escapar. Esto permitía **XSS almacenado** (se ejecutaba código malicioso cada vez que alguien veía la página) y también riesgo de inyección SQL.

Qué hice:

- Usé **prepared statements** para insertar los comentarios de forma segura.
- Apliqué htmlspecialchars(..., ENT_QUOTES, 'UTF-8') al mostrar los textos para evitar que se ejecute código.
- Añadí manejo de errores más limpio y sugerí usar una **CSP** para reforzar la protección.

Por qué es más peligroso el XSS almacenado:
Porque el código malicioso queda guardado en la base de datos y se ejecuta automáticamente para todos los usuarios, no solo para quien hace clic en un enlace.

Prueba rápida:
Antes: al enviar <script>alert('XSS')</script> aparecía una alerta cada vez que se cargaban los comentarios.
Ahora: se muestra el texto literal sin ejecutar nada.

Conclusión:
Con las consultas preparadas y el escape de salida se eliminan los riesgos de inyección y XSS.

EJ 5

Problema detectado:

El código original guardaba las contraseñas en texto plano en la base de datos. Cualquiera con acceso a la BD podía leer todas las contraseñas directamente. También usaba concatenación SQL vulnerable a inyección.

Qué hice:

• Usé password_hash() con bcrypt para hashear las contraseñas antes de guardarlas.

• Implementé prepared statements para prevenir SQL injection.

• Añadí validación: mínimo 8 caracteres con mayúscula, minúscula y número.

• En login se usa password_verify() para comprobar contraseñas de forma segura.

Por qué es crítico almacenar en texto plano:

En una filtración, el atacante obtiene acceso inmediato a todas las credenciales. Como el 65% de usuarios reutiliza contraseñas, puede atacar otros servicios (Gmail, bancos, etc.). Además, incumple el RGPD con multas de hasta €20M.

Plan de migración:

Si ya hay contraseñas en texto plano: añadir columna password_is_hashed y en el login detectar si es texto plano o hash. Si coincide en texto plano, hashear automáticamente en ese momento. Tras 60-90 días, forzar reset a usuarios inactivos.

Prueba rápida:

Antes: password_hash = "admin123" visible.

Ahora: password_hash = "$2y$10$N9qo8uLO..." imposible de revertir.

Conclusión:

Con password_hash() y prepared statements se protegen las credenciales y se previene SQL injection, incluso si la BD es comprometida.