1)

Asumo que el proceso recibe el descriptor del directorio en una variable fd, que se posee una estructura Ext2FSDirEntry que corresponde a la estructura con la que se guardan las entradas de los directorios en sus bloques de datos y que existe una función void load_block(unsigned lba, void* buff) que lee el bloque de dirección lógica lba en buff.

Código:

```
Ext2FSDirEntry entry;
// Leo el directorio "." (el actual) y guardo el número de inodo.
read(fd, &entry, sizeof(Ext2FSDirEntry)); unsigned n = entry.inode_number;
// Avanzo a la siguiente entrada
seek(fd, entry.name_length);
// Leo el directorio ".." (el padre)
read(fd, &entry, sizeof(Ext2FSDirEntry));
// Obtengo el inodo del directorio padre
Ext2FSInode ino = load_inode(entry.inode_number);
// Busco la entrada que corresponde al inodo n
char block[BLOCK_SIZE]; char *name = NULL;
for(int i=0; i<12; ++i) {
    load_block(ino.blocks[i], &block);
    name = buscar_nombre(n, &block);
    if(name != NULL) break;
}
```

```
if (name == NULL) {
    load_block(ino.blocks[12], &block);
    char buff[BLOCK_SIZE];
    for(int i=0; i < BLOCK_SIZE; i+= LBA_SIZE){
        load_block(block[i], &buff);
        name = buscar_nombre(n, &buff);
        if (name != NULL) break;
    }
}
// Buscar de forma análoga para la doble (ino.block[13]) y triple
// (ino.block[14]) indirección.
```

```
char * buscar_nombre(unsigned n, char *block) {
    Ext2FSDirEntry *entry
    unsigned count = 0;
    while (count < BLOCK_SIZE) {
        entry = block;
        if (entry->inode_number == n) break;
        block += entry->record_length;
        count += entry->record_length;
    }
    return count < BLOCK_SIZE ? entry->name : NULL;
}
```

2)

a y b.

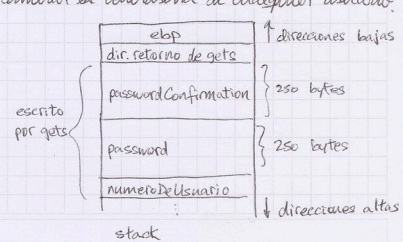
El problema de seguridad está en el uso de gets para leer la confirmación de la contraseña ya que esta función lee caracteres hasta encontrarse con el caracter de end of string y, luego, puede ser manipulado por el atacante para que lea una cantidad arbitraria de caracteres.

El atacante puede hacer buffer overflow, proveer un input que exceda el tamaño del buffer passwordConfirmation para sobreescribir los datos del stack y realizar alguna acción indebida.

Una opción es sobreescribir la dirección de retorno de la función cambiarPassword para saltar a un código malicioso que se escribe en el stack usando también el input de gets. Como la función se ejecuta con permisos de root (o al menos debería), el atacante ganaría acceso total a la máquina. Esto, sin embargo, es difícil si el SO implementa stack randomization.

Otra posibilidad es proveer un input que tenga 500 caracteres que consisten de una cadena de 250 caracteres duplicada, para escribir passwordConfirmation y password con el mismo valor, y caracteres que sobreescriban numeroDeUsuario con el número de cualquier usuario. De esta forma, el atacante podría cambiar la contraseña de cualquier usuario.

Hacer esto es posible ya que las tres variables escritas se encuentran en el stack como muestra la figura de la derecha.



c.

Bob debe poner a root como dueño y grupo del archivo binario que corresponde al programa, darle permiso de lectura y ejecución a todos los usuarios y setear el flag setuid. Este flag hará que cuando un usuario ejecute el programa, el dueño del proceso creado sea el dueño del binario y no el usuario que lo ejecutó. Luego el proceso correrá como root y podrá leer y escribir /etc/passwd. ✓ En realidad Bob ya podía leer y modificar /etc/passw

d.

- Texto plano: puede conocer la contraseña de todos los usuarios y autenticarse como cualquier usuario. ✓
- Hash: no conoce la contraseña pero puede intentar usar fuerza bruta para encontrarla o al menos encontrar una contraseña que tenga el mismo hash, asumiendo que conoce el algoritmo de hash utilizado. ✓ ⊛
- Hash + salt: no podrá obtener la contraseña ya que no conoce el salt con el que fue hasheada. ✓ Igual que pasaría si el salt también es accesible?

⊛ No es necesario conocer la función de hash, podría cambiar su contraseña hasta que el hash de la misma sea igual que un hash objetivo.

3)

a.

i. Conviene intercambiarla lo antes posible para minimizar el tiempo de respuesta del sistema y proveer una mejor experiencia de usuario.

ii. Es mejor almacenar más datos antes de intercambiarlos ya que el tiempo de respuesta no es tan importante en este contexto y porque es más eficiente, tanto para el sistema en cuestión como para toda la red, hacer transferencias de cantidades más grandes de datos ya que se disminuye el overhead de establecer la conexión y, entonces, la contención de la red. Si se agregan varios archivos pequeños a la carpeta y se envía cada uno individualmente, se debe realizar el handshake para establecer la conexión una vez por cada archivo, lo que hace la transferencia más ineficiente y aumenta el uso de la red.

iv. No es conveniente usar un buffer ya que en este contexto de uso el driver suele recibir cantidades suficientes de datos como para hacer el intercambio inmediatamente y el uso de un buffer reduciría la eficiencia (que es muy importante en estas situaciones) ya que se deberían copiar los datos al buffer.

v. Ídem. punto i.

b.

i. No, ya que las operaciones tardan poco y el usuario necesita una respuesta inmediata.

ii. Sí, porque las transferencias pueden demorar mucho y porque, en general, el usuario no necesita que se haga inmediatamente la trans-

ferencia; basta con encolarla con el spooler y que se realice cuando se pueda.

iv. Tiene sentido usar spooling si la GPU recibirá trabajos de varios procesos y si estos no necesitan el resultado del cómputo inmediatamente. De esta forma los procesos quedan libres para realizar otras operaciones en lugar de esperar a que la GPU responda y el spooler mantiene trabajando a la GPU lo más posible.

Esto podría aplicarse al caso del buffer también, solo que contaría para 1 solo proceso

v. Ídem punto i.

c.

i. No, ya que la cantidad de información intercambiada es poca y el CPU es más eficiente que el controlador de DMA para comunicarse con el dispositivo

ii. Sí, porque se transfieren grandes cantidades de datos y se desperdiciarían muchos ciclos del CPU si este se debe encargar de hacer la transferencia. Con DMA, el controlador de DMA puede realizar la transferencia mientras el CPU continúa con otro trabajo

iv. Dado que se suelen hacer cómputos de grandes cantidades de datos, es conveniente dejar al controlador de DMA realizar la transferencia de los datos a computar a la GPU y liberar al CPU, aumentando así el throughput.

v. Ídem punto i.

d. Usaría interrupciones en todos los casos. En i y v porque se debe esperar el input de un usuario que, en términos del CPU, demora mucho, por lo que se desperdiciarían muchos ciclos haciendo polling. En ii y iv porque son operaciones que pueden demorar mucho tiempo en el cual el CPU puede hacer otras cosas.

4)

a. No lo permite ya que no hay forma de determinar qué proceso fue el primero en ofertar realmente con la información provista por el filesystem. Cada proceso crea su archivo con el timestamp de creación de acuerdo a su clock (el de la máquina que ejecuta al proceso), que puede no estar sincronizado con el del resto de los procesos. Luego, es posible que un proceso que no es el primero en ofertar tiene el reloj atrasado tal que al crear su archivo le pone un timestamp anterior al del primer ofertante.

b. El protocolo funciona asumiendo que la red no pierde mensajes y que los procesos no se caen.
Si alguna de estas cosas pudiera suceder, podría ocurrir que alguno de los mensajes de creación de archivo se pierda, ya sea porque se pierde el mensaje en la red o porque el proceso se cae antes de crear el archivo, haciendo que no se pueda completar la confirmación. Esto se conoce como acuerdo bizantino.
Si no se dan ninguno de estos casos, los mensajes llegarán eventualmente y se concretará la confirmación.