LAB2 Sistemas empotrados

Parte 1:

Tarea1: Toolchain de GNU

1 a) man - muestra una vista con información como nombre, sinopsis, descripción, opciones, etc. ej- ‘ man ls ’ devuelve una explicación del comando ls.

size Enlista el tamaño de sección y el tamaño total de los archivos binarios en la lista de argumentos de un fichero, ej- ‘size [-A|-B|-G|--format=compatibility]’

strip - Elimina símbolos u otros datos de ficheros, modifica el fichero en sí, en vez de crear una copia y editarla, ej- ‘strip [-F bfdname |--target=bfdname]’

nm - Muestra los símbolos de un fichero, ej- ‘nm [-A|-o|--print-file-name] [-a|--debug-syms]’

ldd - Imprime los objetos compartidos requeridos por el programa u objeto especificado en la línea de comandos, ej- ‘ldd [option]... file...’

strings - Imprime las secuencias de carácteres imprimibles que una longitud determinada que sean seguidos por un carácter no imprimible, ej- ‘strings [-afovV] [-min-len]’

objdump - Imprime la información de uno o más ficheros, ej- ‘objdump [-a|--archive-headers]’

readelf - Muestra la información de uno o más archivos con formato ELF, ej- ‘readelf [-a|--all]’

objcopy - Copia el contenido de un archivo a otro, ej- ‘objcopy [-F bfdname|--target=bfdname]’

as/gas - Ensambla la salida del compilador gcc , ej- ‘as [-a[cdghlns][=file]] [--alternate] [-D]’

wc - Imprime la cuenta de líneas, palabras y bytes de un fichero, ej- ‘wc [OPTION]... [FILE]...’

free - muestra la memoria utilizada y la disponible en el sistema incluyendo la swap, ej- ‘free -m’ nos muestra la memoria en megabytes

df - Muestra el espacio utilizado y el disponible en todos los sistemas de ficheros montados, ej- ‘df -h’ mostrará la información de manera más entendible para los humanos.

grep/egrep - Busca en uno o más ficheros una cadena determinada de texto, ej- ‘cat /etc/ passwd | grep -i holamundo’ buscará “holamundo” y nos dirá dónde se encuentra.

less - similar al ‘more’ pero permite consultar páginas hacia atrás además de hacia delante, ej- ‘less archivo.c’

more - Se encarga de paginar el texto, mostrando una pantalla cada vez, ej- ‘more archivo.c’ imprimirá el archivo.c página a página.

find - Busca un directorio o fichero en un sistema de ficheros, ej- ‘find /home -name “\*”.ccp | more’

tail - Imprime las diez últimas líneas de un fichero, ej- ‘tail -f -n 20 /var/log/archivo.log’ imprimirá las últimas 20 líneas de registro del fichero archivo.log en directo.

strace - Ejecuta un comando en específico hasta que termine, interpreta y anota los system calls llamados por un proceso y las señales recibidas, ej- ‘strace [-ACdffhikqqrtttTvVwxxyyzZ] [-I n] [-b execve] [-e expr]...’

ld - Combina un número de ficheros, y realoca sus datos, suele ser el último paso en compilar un programa, ej- ‘ld [options] objfile ...’

sudo - Permite a un usuario ejecutar un comando como un superusuario, o como otro usuario con más permisos, ej- ‘sudo, sudoedit — execute a command as another user’

vim|nano|joe - Un editor pequeño, permite editar el código de ficheros además de tener muchas características de otros editores de texto, ej- ‘nano [options] [[+line[,column]] file]...’

md5sum - imprime o verifica las sumas de MD5, ej- ‘md5sum [OPTION]... [FILE]...’

file - Prueba cada argumento para intentar clasificarlo, hace tres pruebas diferentes; filesystem, magic, y language, la primera prueba que se apruebe determinará el tipo de archivo que se imprimirá, ej- ‘file [-bcdEhiklLNnprsSvzZ0] [--apple] [--extension] [--mime-encoding]’

1 b)

1º Ejemplo:

grep -oi “palabra” archivo.txt | wc -l > salida.txt

“grep” es un comando para buscar texto en archivos, “-oi” hace que solo se busque las palabras que coinciden sin tener en cuenta las mayúsculas o minúsculas. Después, con una “pipe”, se conecta el comando “wc -l”, es decir, cuenta el número de palabras encontradas, que es el output de la instrucción anterior.

ls -l | grep “texto”

Este comando se usa para buscar archivos mediante una palabra clave (texto). Consiste en listar todos los archivos en un directorio con “ls -l” y pasarle el output a “grep” con una tubería, el cual se encargará de buscar los archivos cuyo nombre contenga “texto”.

top | grep "Cpu(s)"

“top” muestra la información en tiempo real del sistema y “grep “Cpu(s)” “ filtra los resultados para mostrar solo el uso de CPU.

1 c)

**.bss:**es una instrucción recoge las variables no inicializadas

**.data:** es una instrucción recoge las variables estáticas y globales inicializadas del proceso.

**.rodata:**es igual que la instrucción .data pero esta recoge datos de sólo lectura.

**.noinit:**esta instrucción se utiliza si queremos que una sección de RAM que no se inicialice al inicio, podemos especificar una región dedicada y una sección de salida para ella.

**.text**: es una instrucción que guarda el código máquina del programa que estemos trabajando.

1 d)

El heap es el área de memoria reservada para el almacenamiento de memoria dinámica, manipulada a través de malloc(), realloc(), free(), etc.La pila es módulo donde se guardan las variables locales que por ejemplo se crean en una función.

1e)

En heap se almacena las variables memoria dinámica, aquellas cuyo tamaño no se conoce durante tiempo de ejecución, es decir, se almacenan con memoria dinámica y por lo tanto no se va a saber de su tamaño hasta el momento de ejecución.

Tarea 2

2a) *-Ver código adjunto-*

2b) Las variables estáticas se inicializan en el proceso de carga del programa, las variables automáticas se inicializan cuando se ejecuta la función que las contiene, a menos que se les asigne un valor concreto o se use un constructor para hacerlo. Las variables que se guardan en el heap o en stack no se inicializan automáticamente, se les debe dar un valor tras asignarlas en el heap o stack. Las variables ELF se inicializan en el proceso de carga del programa, y las variables que se definen en el código fuente pero nunca llegan a tener una existencia real no se llegan a inicializar.

2c) No se puede visualizar el nombre de todas. Las variables estáticas son visibles y mantienen sus nombres asignados en el código fuente. Las variables automáticas son visibles si están siendo usadas en el momento del desensamblado. Las variables guardadas en stack, al igual que las automáticas, se pueden ver si están siendo usadas en ese momento, además de ésto, suelen tener nombres generados por el compilador. Las variables guardadas en heap no se pueden ver directamente, ya que sus referencias están almacenadas en punteros. Las variables ELF son visibles pero sus nombres son generados por el compilador. Las variables que se optimizan no llegan a existir durante la compilación, por lo que no son visibles.

2d)

Variable estática - almacena datos fijos, de modo que el valor de la variable no cambia a lo largo de las llamadas de la función.

Variable dinámica - su valor se consulta o resuelve cuando se consultan, son actualizadas por el propio sistema operativo. Se crean/destruyen durante la ejecución de un programa, tienen tipo y valor pero no nombre.

Variable automática - se declaran como una instancia de una clase específica, están enlazadas a la instancia de la clase en la que están definidas, por lo que sólo existen en el entorno local.

2e)

Para poder obtener información de las variables del programa se han usado dos herramientas, readelf y objdump, los cuales no brindan información de todas las variables ya que para eso se debería usar un depurador en tiempo real como gdb.

readelf -s --wide tarea2

objdump -d tarea2

De estas instrucciones se puede extraer que las variables ELF\_rows y ELF\_cols, ambas globales, pertenecen al al archivo elf.

2f)

Fragmento de compilación de forma cruzada:

00000000000009b0 <\_init>:

9b0: d503201f nop

9b4: a9bf7bfd stp x29, x30, [sp, #-16]!

9b8: 910003fd mov x29, sp

9bc: 9400005e bl b34 <call\_weak\_fn>

9c0: a8c17bfd ldp x29, x30, [sp], #16

9c4: d65f03c0 ret

Desensamblado de la sección .plt:

00000000000009d0 <.plt>:

9d0: a9bf7bf0 stp x16, x30, [sp, #-16]!

9d4: b0000090 adrp x16, 11000 <\_FRAME\_END\_+0xff7c>

9d8: f9479e11 ldr x17, [x16, #3896]

9dc: 913ce210 add x16, x16, #0xf38

9e0: d61f0220 br x17

9e4: d503201f nop

9e8: d503201f nop

9ec: d503201f nop

Fragmento de compilación normal:

000000000001000 <\_init>:

1000: f3 0f 1e fa endbr64

1004: 48 83 ec 08 sub $0x8,%rsp

1008: 48 8b 05 d9 2f 00 00 mov 0x2fd9(%rip),%rax # 3fe8 <\_gmon\_start\_@Base>

100f: 48 85 c0 test %rax,%rax

1012: 74 02 je 1016 <\_init+0x16>

1014: ff d0 call \*%rax

1016: 48 83 c4 08 add $0x8,%rsp

101a: c3 ret

Desensamblado de la sección .plt:

0000000000001020 <.plt>:

1020: ff 35 42 2f 00 00 push 0x2f42(%rip) # 3f68 <GLOBAL\_OFFSET\_TABLE+0x8>

1026: f2 ff 25 43 2f 00 00 bnd jmp \*0x2f43(%rip) # 3f70 <GLOBAL\_OFFSET\_TABLE+0x10>

102d: 0f 1f 00 nopl (%rax)

1030: f3 0f 1e fa endbr64

1034: 68 00 00 00 00 push $0x0

1039: f2 e9 e1 ff ff ff bnd jmp 1020 <\_init+0x20>

103f: 90 nop

1040: f3 0f 1e fa endbr64

1044: 68 01 00 00 00 push $0x1

1049: f2 e9 d1 ff ff ff bnd jmp 1020 <\_init+0x20>

104f: 90 nop

1050: f3 0f 1e fa endbr64

1054: 68 02 00 00 00 push $0x2

1059: f2 e9 c1 ff ff ff bnd jmp 1020 <\_init+0x20>

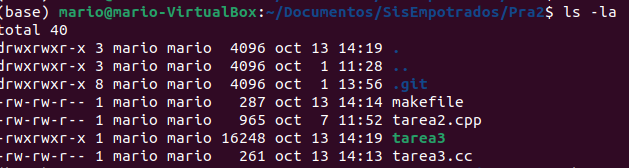
105f: 90 nop

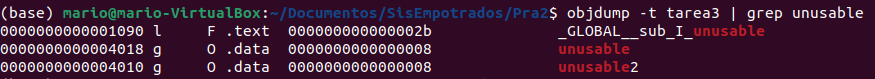
1060: f3 0f 1e fa endbr64

2g)

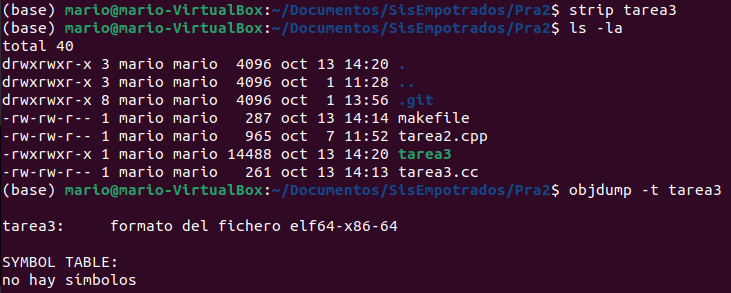
Tarea 3:

3a)



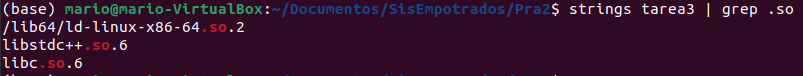


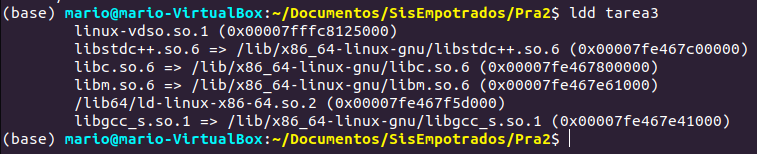
Una vez creado el ejecutable (archivo elf), lo examinamos y observamos que este contiene un montón de símbolos. Una vez aplicado el strip, se puede ver como el tamaño del ejecutable se ha reducido y a través de un objdump se observa que es por la eliminación de todos los símbolos.



Es por esto que para programas que se ejecutan en dispositivos más o menos potentes la diferencia no es perceptible, pero si se habla de sistemas empotrados donde los recursos son más limitados, puede tener una diferencia notable.

3b)



Se puede observar que las dependencias dinámicas usadas por el programa son:

linux-vdso.so.1

libstdc++.so.6

libc.so.6

libm.so.6

libgcc\_s.so.1

3c)

-3.1

ARM

La función askDim empieza en la dirección c6c y ocupa 56

La función makeMatriz empieza en la dirección ca8 y ocupa 324

La función matCalc empieza en la dirección df0 y ocupa 52

NORMAL

La función askDim empieza en la dirección 12f2 y ocupa 60

La función makeMatriz empieza en la dirección 132f y ocupa 278

La función matCalc empieza en la dirección 1446 y ocupa 45

-3.2

Como el main ocupa 165 con las funciones desarrolladas ocupan un 68%

del formato .text.

-3.3

En gcc -S genera el código ensamblador a partir del código fuente, mientras que objdump -j .text muestra el código ensamblador a partir del archivo objeto.

-3.4

-o1: es igual al modo predeterminado(o)

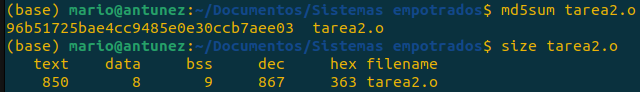
-o3: es el nivel de optimización más rápido





-ofast es el mismo nivel de eficiencia que o3 solo que aprovecha un extra de optimización pero sacando algunas normas si se puede para optimizar más: 

-og sirve para optimizar la experiencia de depuración:



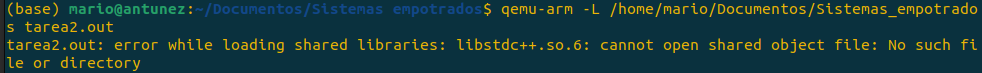
3b) Para crear un binario estático hay que añadir -static al comando de compilación. El ldd no hace nada al ser un ejecutable estático, ldd sólo muestra las dependencias estáticas.

3c)

Para crear la versión hexadecimal, hay que usar: objcopy -0 ihex tarea2ARM.out tarea2HEX.hex

**Tarea 4:**

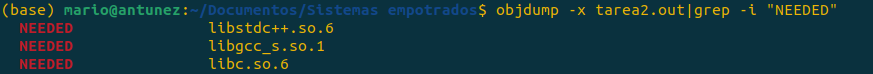
Inicialmente compilando el archivo de forma SO less en la terminal expone el siguiente error:



El error “cannot open shared object file: No such file or directory” se produce cuando el sistema no puede encontrar la biblioteca compartida libstdc++.so.6 en el sistema anfitrión. Esta biblioteca es necesaria para ejecutar binarios ELF en sistemas Linux.

Esta librería es una biblioteca compartida crítica que implementa GLIBCXX\_3.4.20 y otras versiones. Se utiliza en muchos compiladores y aplicaciones construidas con C++. Contiene funciones, clases y algoritmos comúnmente utilizados para contenedores, iteradores.

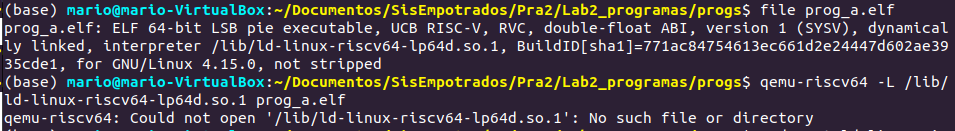
Después escribiendo el comando con el formato siguiente objdump -x |grep -i "NEEDED” salta esto:



**Tarea 5:**

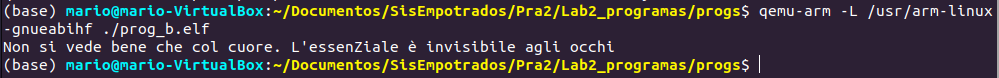
Primero se tiene que averiguar para qué arquitectura está compilado cada elf, para ello se pueden usar varios comandos, el más fácil “file prog\_x.elf”

prog\_a.elf: RISC-V

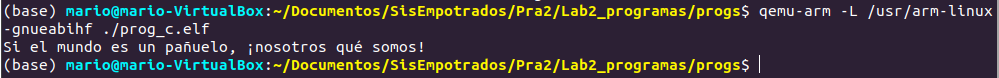


El programa “a” no se puede ejecutar.

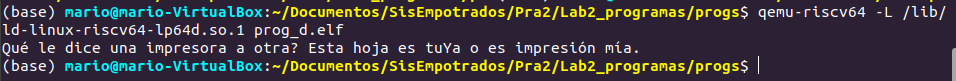
prog\_b.elf: ARM



prog\_c.elf: ARM



prog\_d.elf: RISC-V



**Tarea 6:**

6.1)

Tal como se puede apreciar en la imagen la ISA de este archivo es tipo ARM

6.2)Aplicamos para el primero este comando

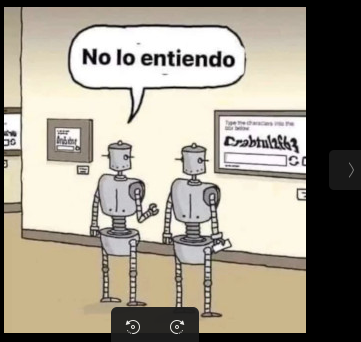


El conv=notrunc es el formato de seguridad para no fastidiar el disco duro en caso de error

El skip es 11048 - ((11040 - 1040) esta resta es para poner lo del .data ya que esto empieza a leer desde ahí).

Blob nos convertirá los datos los volcará en una imagen

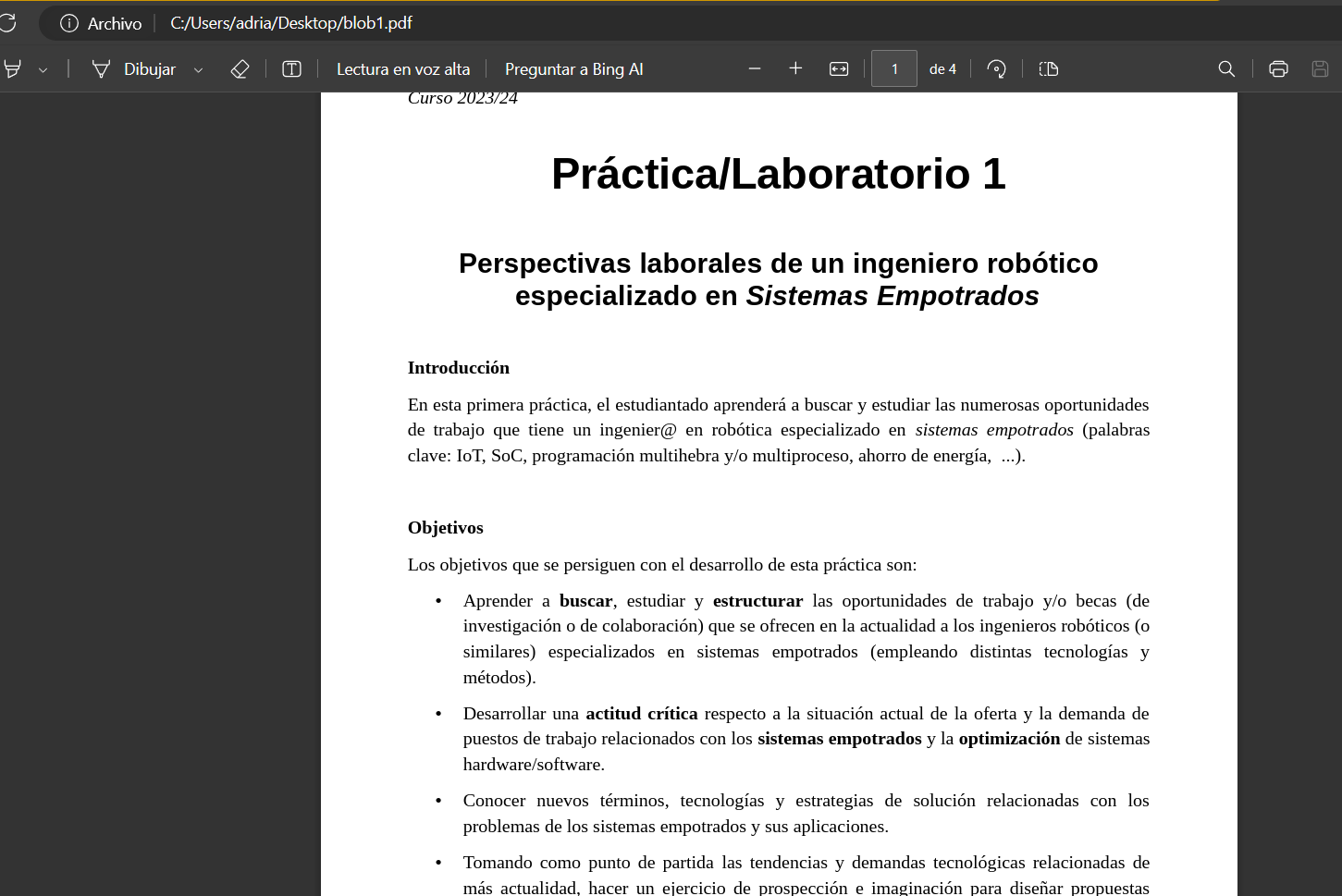
Y sale esta imagen



El otro con el siguiente comando

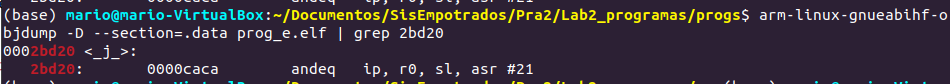


nos deja este pdf:



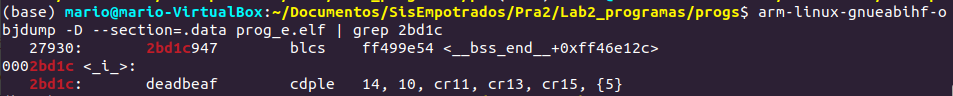
es el pdf de la práctica. 1

6.3)



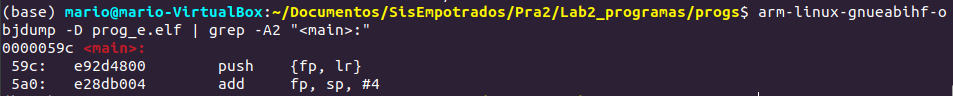
Se puede observar que Paquito no come muy saludable, ¡porque come caca!. Es importante remarcar que al ser little endian se leería al revés.

6.4)



Peter tampoco come sano.

6.5)



La segunda instrucción es una suma y está codificada como 400bd82e (little endian)

**Tarea 7:**

Para depurar de forma cruzada hay que escribir:

qemu-arm -g 1233 -L / usr/arm-linux-gnueabihf/ tarea2ARM

Ésto sería para el puerto TCP 1233, y no imprime nada. Luego hay que abrir otra terminal e iniciar el depurador gdb para ARM, luego hay que poner el comando:

target remote localhost:<puerto>

arm-none-eabi-gdb tarea2ARM

Ahora se ha establecido un breakpoint en la sección \_start() en vez de en la función main porque es la primera función que está en la sección al hacer el objdump. Si ahora depuramos en remoto y usamos el comando ‘s’ para que dé un paso devolverá un error, porque la sección \_start no tiene información que gdb pueda leer. Para evitar esto hay que poner un breakpoint en el main usando ‘b main’, y usar ‘c’ para que avance hasta el principio y pueda empezar de manera normal.