

PRÁCTICA 2 – SISTEMAS EMPOTRADOS

Entrenamiento con el toolchain de GNU y QEMU para sistemas empotrados

Lucas Serrano Jiménez

César San Blas Leal

TAREA 1

1a) Explique brevemente y con ejemplo que hace cada una de las ordenes anteriores y las que le indique (en su caso) el profesor de practicas.

- **man**: muestra ayuda sobre otros comandos, incluyendo una descripción de sus argumentos.

```
mkdir [OPTION]... DIRECTORY...

DESCRIPTION
  Create the DIRECTORY(ies), if they do not already exist.

  Mandatory arguments to long options are mandatory for short options
  too.

  -m, --mode=MODE
        set file mode (as in chmod), not a=rwx - umask

  -p, --parents
        no error if existing, make parent directories as needed

  -v, --verbose
        print a message for each created directory

  -Z
        set SELinux security context of each created directory to the
        default type

  --context[=CTX]
        like -Z, or if CTX is specified then set the SELinux or SMACK
        security context to CTX

Manual page mkdir(1) line 7/55 51% (press h for help or q to quit)
```

- **size**: muestra el tamaño de un programa en decimal y hexadecimal, además del tamaño del texto en número de caracteres, los datos y las variables no usadas.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ size HW
text      data      bss      dec      hex filename
1376      600         8     1984     7c0 HW
```

- **strip**: elimina información de depuración del ejecutable.
- **nm**: lista los símbolos de un archivo ejecutable.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ nm HW
000000000000038c r __abi_tag
00000000000004010 B __bss_start
00000000000004010 b completed.0
                                w __cxa_finalize@GLIBC_2.2.5
00000000000004000 D __data_start
00000000000004000 W data_start
00000000000001090 t deregister_tm_clones
00000000000001100 t __do_global_ctors_aux
00000000000003dc0 d __do_global_ctors_aux_fini_array_entry
00000000000004008 D __dso_handle
00000000000003dc8 d _DYNAMIC
00000000000004010 D _edata
00000000000004018 B _end
```

- **ldd**: muestra dependencias compartidas de un programa.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ ldd HW
linux-vdso.so.1 (0x00007ffc560cb000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f87b1e00000)
/lib64/ld-linux-x86-64.so.2 (0x00007f87b20d1000)
```

- **strings**: muestra las cadenas de texto legibles de un archivo ejecutable.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ strings HW
/lib64/ld-linux-x86-64.so.2
__cxa_finalize
__libc_start_main
puts
libc.so.6
GLIBC_2.2.5
GLIBC_2.34
_ITM_deregisterTMCloneTable
__gmon_start__
_ITM_registerTMCloneTable
PTE1
u+UH
Hello, World!
```

- **objdump**: proporciona diversos tipos de información detallada sobre un archivo ejecutable.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ objdump HW
Usage: objdump <option(s)> <file(s)>
Display information from object <file(s)>.
At least one of the following switches must be given:
-a, --archive-headers    Display archive header information
-f, --file-headers       Display the contents of the overall file header
-p, --private-headers    Display object format specific file header contents
-P, --private=OPT,OPT... Display object format specific contents
-h, --[section-]headers  Display the contents of the section headers
-x, --all-headers        Display the contents of all headers
-d, --disassemble        Display assembler contents of executable sections
```

Por ejemplo, con -a se obtendría información acerca del título y el tipo de formato del archivo.


```
(base) cesar@cesar-virtual-machine:~/Desktop$ objdump -a HW
HW:      file format elf64-x86-64
HW
```

- **readelf**: muestra información acerca de la estructura de un archivo ejecutable.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ readelf HW
Usage: readelf <option(s)> elf-file(s)
Display information about the contents of ELF format files
Options are:
  -a --all                Equivalent to: -h -l -S -s -r -d -V -A -I
  -h --file-header        Display the ELF file header
  -l --program-headers    Display the program headers
  --segments              An alias for --program-headers
  -S --section-headers    Display the sections' header
  --sections              An alias for --section-headers
```

- **objcopy:** crea un nuevo archivo ejecutable copiando el proporcionado como primer argumento.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ objcopy HW HW_copy
(base) cesar@cesar-virtual-machine:~/Desktop$
```



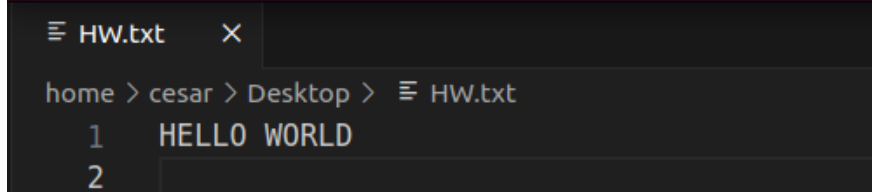
- **as/gas:** ensambla y genera el código objeto de un programa en ensamblador (.s).

```
NAME
    AS - the portable GNU assembler.

SYNOPSIS
    as [-a[cdghlns][=file]] [--alternate] [-D]
       [--compress-debug-sections] [--nocompress-debug-sections]
       [--debug-prefix-map old=new]
       [--defsym sym=val] [-f] [-g] [--gstabs]
```

- **wc:** muestra las líneas, palabras y caracteres, en ese orden, que contiene un archivo de texto.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ wc HW.txt
1  2 12 HW.txt
(base) cesar@cesar-virtual-machine:~/Desktop$
```



- **free:** muestra la cantidad de memoria libre y ocupada del sistema, en kibibytes.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ free
              total        used         free       shared    buff/cache   available
Mem:      3964440      1127804       588144        35264       2248492       2544700
Swap:      2191356         2840       2188516
```

Usando los argumentos adicionales -h y --si, se obtiene una lectura más comprensible.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ free -h --si
```

	total	used	free	shared	buff/cache	available
Mem:	3,9G	1,1G	574M	34M	2,2G	2,5G
Swap:	2,1G	2,0M	2,1G			

- **file:** te da información sobre un archivo como su tipo y contenido.

```
lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados$ file main.c
main.c: C source, ASCII text
```

- **md5sum:** Proporciona una secuencia alfanumérica para un archivo que cambia con cada modificación que se le haga, de esta manera se pueden comprobar que las instalaciones han ido correctamente o si se ha modificado el contenido.

```
lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados$ md5sum main.c
2feb2a8fc094ddf7d7c7a907af21c8c0 main.c
lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados$ nano main.c
lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados$ md5sum main.c
d7ceabf48c5196185ba7efacbc2b5238 main.c
```

- **vim|nano|joe:** son 3 editores de texto integrados en el entorno de la terminal, sirven para programar, entre otras cosas.

- Vim:

```
int main(){
    printf("hola mundo");
}
```

Parte 1

Tarea 1: Teclado de GNU

El profesor impartirá un breve tutorial que abarcará entre otras cosas el estudio de la estructura y funcionamiento básico de un sistema GNU/Linux. Una (funcionamiento y ordenes básicas, estructura del sistema de archivos, directorios principales, manejo de la consola, permisos, etc.) y la estructura de un buen IDE.

En el tutorial se le mostrará el funcionamiento de algunas ordenes GNU fundamentales para

-- INSERTAR -- 1,1 Todo

```
Nano:
GNU nano 4.8                               main.c
#include <stdio.h>

int main(){
    printf("hola mundo");
}

[ 5 líneas leídas ]
^G Ver ayuda ^O Guardar ^W Buscar ^K Cortar Texto ^J Justificar ^C Posición
^X Salir ^R Leer fich. ^\ Reemplazar ^U Pegar ^T Ortografía ^_ Ir a línea
```

```
Joe:
I A main.c (c) Row 1 Col 1
#include <stdio.h>

int main(){
    printf("hola mundo");
}

as empujados como: man, size, strip, nm, ldd, strings, objdump, readelf,
objcopy, as/objcopy, wc, free, df, grep/egrep, less, more, find, tail, strace, ld,
Joe's Own Editor 4.6 (utf-8) ** Type Ctrl-K Q to exit or Ctrl-K H for help **
```

- **sudo:** otorga privilegios de administrador al usuario para la ejecución de ese comando.
- **Id:** sirve para enlazar varios archivos objeto a un archivo ejecutable. De esta manera un programa puede, por ejemplo, ejecutar funciones que estén desarrolladas en otros archivos de código fuente.
- **strace:** Muestra las llamadas al sistema que realiza un programa, como las impresiones por pantalla o las reservas de memoria para las variables.
- **tail:** Muestra por pantalla las últimas líneas (por defecto 10) de un archivo de texto, también tiene la opción de ir mostrando sus actualizaciones.

```

lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados$ tail lineas.txt
linea 1
linea 2
linea 3
linea 4
linea 5
linea 6
linea 7
linea 8
linea 9
linea 10

```

- **find:** Sirve para buscar archivos y directorios del sistema.
- **more:** Permite al usuario controlar como se muestra el texto por pantalla, pudiendo avanzar pulsando la barra espaciadora.
- **less:** Muestra por pantalla el contenido del archivo indicado.
- **grep/egrep:** Busca en uno o varios archivos líneas que contengan la string indicada por consola.
- **df:** Se utiliza para conocer información sobre el espacio libre en el sistema, la opción -h lo hace más cómodo de leer.

```

lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados$ df -h
S.ficheros      Tamaño Usados  Disp Uso% Montado en
udev            5,5G    0      5,5G  0% /dev
tmpfs           1,2G    2,0M    1,2G  1% /run
/dev/nvme0n1p4  63G     50G     9,4G  85% /
tmpfs           5,6G    23M     5,5G  1% /dev/shm
tmpfs           5,0M    4,0K     5,0M  1% /run/lock
tmpfs           5,6G    0       5,6G  0% /sys/fs/cgroup
/dev/loop0      56M     56M     0     100% /snap/core18/2721
/dev/loop2      64M     64M     0     100% /snap/core20/1852

```

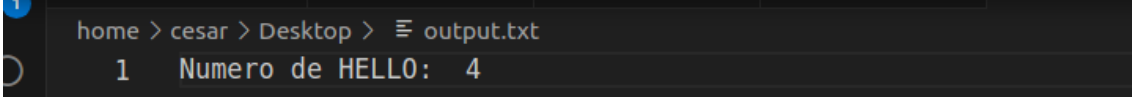
1b) Incluye

al menos 3 ejemplos ilustrativos donde use varias de estas órdenes y las herramientas de redirección y tuberías (pipes) incluidas en todo Linux/Unix. Ayuda: Por ejemplo, ¿cómo resolvería esto concatenando órdenes Linux/Unix en una sola línea? → “Quiero saber cuántas veces aparece una cierta palabra en un archivo de texto (sin tener en cuenta mayúsculas y minúsculas) y guardar el resultado en el archivo salida.txt”.

Esta concatenación de comandos cuenta el número de veces que aparece la palabra indicada como argumento en el archivo de texto dado, ordenando cada aparición en una línea y contando después la cantidad de estas. El resultado lo imprime en un archivo de texto junto con una etiqueta.

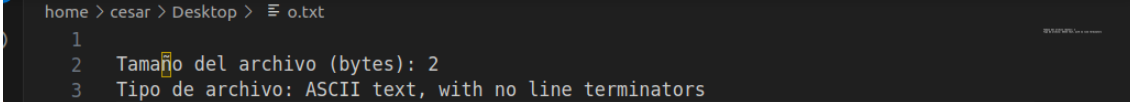
En un primer momento se imprimían en líneas diferentes así que se ha utilizado un comando adicional que sustituye el salto de línea por un espacio en un nuevo fichero. Finalmente, se borra el primer fichero generado y se guarda únicamente el que interesa.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ echo "Numero de HELLO: " >> counter.txt | grep -o "HELLO" HW.txt | wc -l >> counter.txt && tr '\n' ' ' < counter.txt > output.txt | rm counter.txt
(base) cesar@cesar-virtual-machine:~/Desktop$
```



Este segundo ejemplo consiste en mandar a un archivo .txt la información de otro archivo (también .txt en este caso). En lugar del método anterior, se ha utilizado *printf* para imprimir en el archivo de destino con el formato deseado. De esta manera resulta mucho más sencillo manejar los saltos de línea y no hace falta trabajar con archivos temporales.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ printf "Tipo de archivo: " >> o.txt | file -b output.txt >> o.txt | printf "\nTamaño del archivo (bytes): " >> o.txt | stat -c "%s" output.txt >> o.txt
(base) cesar@cesar-virtual-machine:~/Desktop$
```



1c) En el contexto de un archivo ELF, ¿qué son y para que se utilizan las secciones .bss, .data, .rodata, .noinit, .text?

.bss: La sección .bss no ocupa espacio en el archivo .ELF. En esta sección se encuentran las variables que se inician en tiempo de ejecución, de modo que al cargar el programa en memoria se les reserva espacio en la sección .bss y no en la sección de datos.

.data: La sección .data sí ocupa espacio en el archivo .ELF y almacena las variables inicializadas. Durante la ejecución del programa estos valores se copian al espacio de memoria que corresponda.

.rodata: La sección .rodata corresponde a valores constantes dentro del programa, que no serán modificados. Estos datos son invariables durante el tiempo de ejecución.

.noinit: almacena datos que no se inician automáticamente al cargar el programa en memoria. Esta sección es útil para datos que deben conservar su valor entre reinicios o que no necesitan inicialización inmediata.

.text: almacena el código ejecutable del programa, es decir, las instrucciones de máquina que el procesador ejecutará para realizar las operaciones del programa, determinando su lógica y comportamiento.

1d) En el contexto de la ejecución de un programa, explique la diferencia entre la pila (stack) y el montículo (heap).

Estos dos conceptos, se refieren a áreas de la memoria utilizadas para almacenar datos en tiempo de ejecución, pero tienen distinta finalidad y características. Mientras que el stack está destinada a la gestión de datos relacionados con la ejecución de funciones, como las variables locales o direcciones de retorno, el heap es una estructura dinámica compartida por todos los

subprocesos utilizada para almacenar datos en ejecución cuyo tamaño o duración no se conoce en tiempo de compilación.

En cuanto a manejo y eficiencia, el stack libera la memoria de forma automática al salir del contexto de la función con una gestión rápida y eficiente. Por el contrario, en el heap la liberación de memoria debe explicitarse por el programador (aunque algunos lenguajes como C# constan de su propio recolector de basura) y el acceso a esta memoria es más lento.

1e) ¿Por que no existe una seccion de tamano definido para el heap?

El heap es la zona de memoria dedicada a almacenar datos en tiempo de ejecución. Es decir, es donde se reservan las zonas de memoria dinámica. Por este motivo no sería coherente que tuviera un tamaño definido ya que en tiempo de ejecución podría verse superado o desaprovechado en función de los requisitos del programa.

Al no tener un tamaño definido para el heap, los programas pueden solicitar y liberar memoria según lo necesiten y conseguir una gestión de la memoria más eficiente.

TAREA 2

2a) Diseñe un programa sencillo en C o C++ no interactivo y que realice una tarea sencilla. No use sólo la función obligatoria main para implementar su programa sino varias llamadas a funciones de su invención. Una vez escrito y comprobado el programa, realice las modificaciones oportunas para asegurar que:

- Su programa use tanto variables estáticas como automáticas (prefijo: st_, auto_).
- Algunas variables: Figura 2: Él usaba C++ para oscurecer el código. Buscad C++ mangling...
 - se guarden en la pila (prefijo: stack_) y otras en el montículo o heap (prefijo: heap_)
 - otras en el mismo ejecutable (prefijo: ELF_)
 - otras se definen en el código fuente pero no llegan nunca a tener existencia real (se optimizan). (prefijo: rperez_)
- Renombre cada variable y constante según el sitio donde piensen que se encontrará en ejecución. Ej. heap_contador, ELF_anchura, stack_i, rperez_j.

// Este programa ordena una matriz 3x3 y devuelve la suma de todos sus elementos.

```
#include <stdio.h>
```

```
#include <stdint.h>
```

```
const int A = 7;
```



```
void swap(int *stack_a, int *stack_b) {  
    int stack_temp = *stack_a;  
    *stack_a = *stack_b;  
    *stack_b = stack_temp;  
}
```

```
void sortMatrix(int stack_mat[3][3]) {  
    int stack_tempArr[9];  
    int stack_k = 0;  
  
    for (int heap_i = 0; heap_i < 3; heap_i++) {  
        for (int heap_j = 0; heap_j < 3; heap_j++) {  
            stack_tempArr[stack_k++] = stack_mat[heap_i][heap_j];  
        }  
    }  
  
    for (int heap_i = 0; heap_i < 9; heap_i++) {  
        for (int heap_j = 0; heap_j < 8; heap_j++) {  
            if (stack_tempArr[heap_j] > stack_tempArr[heap_j + 1]) {  
                swap(&stack_tempArr[heap_j], &stack_tempArr[heap_j + 1]);  
            }  
        }  
    }  
  
    stack_k = 0;  
  
    for (int heap_i = 0; heap_i < 3; heap_i++) {  
        for (int heap_j = 0; heap_j < 3; heap_j++) {  
            stack_mat[heap_i][heap_j] = stack_tempArr[stack_k++];  
        }  
    }  
}
```

```
}
```

```
int8_t suma(int stack_mat[3][3]){  
    int8_t stack_sum = 0;  
    for (int heap_i = 0; heap_i < 3; heap_i++) {  
        for (int heap_j = 0; heap_j < 3; heap_j++) {  
            stack_sum+=stack_mat[heap_i][heap_j];  
        }  
    }  
    return stack_sum;  
}
```

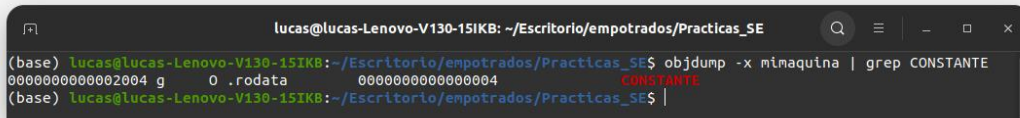
```
int main() {  
    int ELF_st_matrix[3][3] = {  
        {4, 5, 7},  
        {8, 1, 6},  
        {3, 2, 9}  
    };  
    auto ELF_auto_sumaContenido = 0;  
    long st_rperez_a;  
  
    sortMatrix(ELF_st_matrix);  
    ELF_auto_sumaContenido = suma(ELF_st_matrix);  
    printf("%d \n",ELF_auto_sumaContenido);  
  
    return 0;  
}
```

2b) Algunas variables se inicializarán en el proceso de carga del programa o al comienzo de éste y otras no → ¿cuáles?

En el proceso de carga se inicializarán las variables globales o las explícitamente declaradas en el código. El resto se inicializan en tiempo de ejecución y, dependiendo de su declaración, puede ser con un valor aleatorio o no.

2c) ¿Puede visualizar el nombre de todas las variables una vez desensamblado el programa? ¿Por qué? (Ayuda: algunas estarán “apiladas”)

Con la siguiente concatenación de comandos se pueden visualizar las variables estáticas del programa:



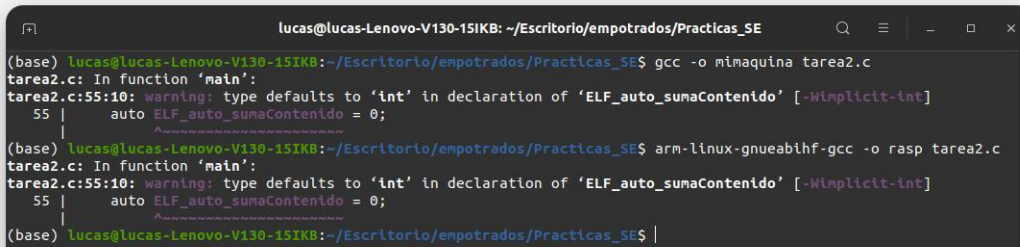
```
lucas@lucas-Lenovo-V130-15IKB: ~/Escritorio/empotrados/Practicas_SE
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$ objdump -x mimquina | grep CONSTANTE
0000000000002004 g      0 .rodata          0000000000000004      CONStANTE
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$
```

2d) Explique cuál es la diferencia entre variable estática, dinámica y automática.

La diferencia reside en su ciclo de vida. Una variable estática dura tanto como dure la ejecución del programa y mantiene su valor. Una variable dinámica se crea en tiempo de ejecución y su ciclo de vida es controlado por el programa en cuestión, pudiendo eliminarse antes de que termine. Finalmente, una variable automática está dentro del ámbito de una función, se crea al llamarla y se destruye al terminar la ejecución de la función.

2f) Compile el programa de forma nativa para el microprocesador del laboratorio y de forma cruzada para el microprocesador de la Raspberry Pi 3 Model 3 y compatibles. Muestre cómo el código ensamblador que se produce es distinto.

La siguiente imagen corresponde a la compilación, tanto nativa como cruzada, del programa expuesto en el apartado a. Primero se realiza la compilación nativa para el sistema utilizado (ordenador con Ubuntu 20.04) y después, utilizando qemu, para el sistema ARM de la raspberry 3.



```
lucas@lucas-Lenovo-V130-15IKB: ~/Escritorio/empotrados/Practicas_SE
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$ gcc -o mimquina tarea2.c
tarea2.c: In function 'main':
tarea2.c:55:10: warning: type defaults to 'int' in declaration of 'ELF_auto_sumaContenido' [-Wimplicit-int]
   55 |     auto ELF_auto_sumaContenido = 0;
      |
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$ arm-linux-gnueabihf-gcc -o rasp tarea2.c
tarea2.c: In function 'main':
tarea2.c:55:10: warning: type defaults to 'int' in declaration of 'ELF_auto_sumaContenido' [-Wimplicit-int]
   55 |     auto ELF_auto_sumaContenido = 0;
      |
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$
```

A continuación, parte del código ensamblador perteneciente a la función *sortMatrix()* de los ejecutables resultantes. Primero el compilado nativamente, seguido del compilado de forma cruzada.

```

lucas@lucas-Lenovo-V130-15IKB: ~/Escritorio/empotrados/Practicas_SE
000000000000119a: <sortMatrix>:
119a: f3 0f 1e fa      endbr64
119e: 55              push    %rbp
119f: 48 89 e5         mov     %rsp,%rbp
11a2: 48 83 ec 00      sub     $0x00,%rsp
11a6: 48 89 7d a8      mov     %rdi,%rax
11aa: 64 48 8b 04 25 28 00 mov     %fs:0x28,%rax
11b1: 00 00
11b3: 48 89 45 f8      mov     %rax,%rax
11b7: 31 c0           xor     %eax,%eax
11b9: c7 45 b4 00 00 00 movl    $0x0,-0x4c(%rbp)
11c0: c7 45 b8 00 00 00 movl    $0x0,-0x48(%rbp)
11c7: eb 4d           jmp     1216 <sortMatrix+0x7c>
11c9: c7 45 bc 00 00 00 movl    $0x0,-0x44(%rbp)
11d0: eb 3a           jmp     120c <sortMatrix+0x72>
11d2: 8b 45 b8         mov     -0x48(%rbp),%eax
11d5: 48 63 d8         movslq  %eax,%rdx
11d8: 48 89 d8         mov     %rdx,%rax
11db: 48 01 c0         add     %rax,%rax
11de: 48 01 d8         add     %rdx,%rax
11e1: 48 c1 e0 02      shl     $0x2,%rax
11e5: 48 89 c2         mov     %rax,%rdx
11e8: 48 8b 45 a8      mov     -0x58(%rbp),%rax
11ec: 48 0d 0c 02      lea     (%rdx,%rax,1),%rcx
11f0: 8b 45 b4         mov     -0x4c(%rbp),%eax
11f3: 8d 50 01         lea     0x1(%rax),%edx
11f6: 89 55 b4         mov     %edx,-0x4c(%rbp)
11f9: 8b 55 bc         mov     -0x44(%rbp),%edx
11fc: 48 63 d2         movslq  %edx,%rdx
11ff: 8b 14 91         mov     (%rcx,%rdx,4),%edx
1202: 48 90           cltq
1204: 89 54 85 d0      mov     %edx,-0x30(%rbp,%rax,4)
1208: 83 45 bc 01      addl    $0x1,-0x44(%rbp)
120c: 83 7d bc 02      cmpl    $0x2,-0x44(%rbp)
1210: 7e c9           jle     11d2 <sortMatrix+0x38>
1212: 83 45 b8 01      addl    $0x1,-0x48(%rbp)
1216: 83 7d b8 02      cmpl    $0x2,-0x48(%rbp)
121a: 7e ad           jle     11c9 <sortMatrix+0x2f>
121c: c7 45 c0 00 00 00 movl    $0x0,-0x40(%rbp)
1223: eb 60           jmp     1285 <sortMatrix+0xeb>

```

```

lucas@lucas-Lenovo-V130-15IKB: ~/Escritorio/empotrados/Practicas_SE
0000005c: <sortMatrix>:
5c: b580          push    {r7, lr}
5d: b094          sub     sp, #0 ; 0x50
5d2: af00          add     r7, sp, #0
5d4: 6078          str     r0, [r7, #4]
5d6: 4a4a          ldr     r2, [pc, #296] ; (700 <sortMatrix+0x132>)
5d8: 447a          add     r2, pc
5da: 4b4a          ldr     r3, [pc, #296] ; (704 <sortMatrix+0x136>)
5dc: 58d3          ldr     r3, [r2, r3]
5de: 681b          ldr     r3, [r3, #0]
5e0: 64fb          str     r3, [r7, #76] ; 0x4c
5e2: f04f 0300     mov.w   r3, #0
5e6: 2300          movs    r3, #0
5e8: 60fb          str     r3, [r7, #12]
5ea: 2300          movs    r3, #0
5ec: 613b          str     r3, [r7, #16]
5ee: e01d          b.n     02c <sortMatrix+0x5e>
5f0: 2300          movs    r3, #0
5f2: 617b          str     r3, [r7, #20]
5f4: e014          b.n     020 <sortMatrix+0x52>
5f6: 693b          ldr     r3, [r7, #16]
5f8: 220c          movs    r2, #12
5fa: fb02 f303     mul.w   r3, r2, r3
5fe: 687a          ldr     r2, [r7, #4]
600: 441a          add     r2, r3
602: 68fb          ldr     r3, [r7, #12]
604: 1c59          adds    r1, r3, #1
606: 60f9          str     r1, [r7, #12]
608: 6979          ldr     r1, [r7, #20]
60a: fb52 2021     ldr.w   r2, [r2, r1, lsl #2]
60c: 00b0          lsls    r3, r3, #2
610: f107 0150     add.w   r1, r7, #00 ; 0x50
614: 440b          add     r3, r1
616: fb43 2c28     str.w   r2, [r3, #-40]
61a: 697b          ldr     r3, [r7, #20]
61c: 3301          adds    r3, #1
61e: 617b          str     r3, [r7, #20]
620: 697b          ldr     r3, [r7, #20]
622: 2b02          cmp     r3, #2
624: d0e7          ble.n   5f6 <sortMatrix+0x28>
626: 693b          ldr     r3, [r7, #16]

```

TAREA 3

3a) Explique qué sucede al ejecutar `size` y `strip` con un determinado archivo ELF. Elabore un programa (el que quiera) en el que la orden `strip` tenga un efecto evidente. ¿Qué sentido tiene esta orden en un sistema empotrado (es de hecho una orden bastante importante)?

La orden *strip* se usa para eliminar información innecesaria de archivos binarios con el objetivo de reducir el tamaño y mejorar su eficiencia en sistemas de recursos limitados. En el ejemplo siguiente, tras usar este comando, se reduce el tamaño del archivo de 16136 a 14472 bytes. Sin embargo, mediante *size* se comprueba que el tamaño de las secciones no se ha alterado, por lo

que, como se verifica a continuación, el código sigue siendo completamente funcional y se ejecuta correctamente.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ ls -l tarea2
-rwxrwxr-x 1 cesar cesar 16136 oct 20 13:04 tarea2
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ size tarea2
   text    data     bss      dec     hex filename
   2284     608        8    2900    b54 tarea2
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ strip tarea2
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ size tarea2
   text    data     bss      dec     hex filename
   2284     608        8    2900    b54 tarea2
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ ls -l tarea2
-rwxrwxr-x 1 cesar cesar 14472 oct 20 13:05 tarea2
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ ./tarea2
45
```

De forma más dirigida y agresiva, se puede usar *strip* para eliminar secciones específicas de un archivo binario, como se observa a continuación.

```
lucas@lucas-Lenovo-V130-15IKB: ~/Escritorio/empotrados/Practicas_SE
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$ objdump -j .rodata -t programa
programa:      formato del fichero elf64-x86-64

SYMBOL TABLE:
0000000000002000 l   d  .rodata      0000000000000000      .rodata
0000000000002000 g   O  .rodata      0000000000000004      _IO_stdin_used
0000000000002004 g   O  .rodata      0000000000000004      A

(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$ strip -R .rodata programa
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$ objdump -j .rodata -t programa
programa:      formato del fichero elf64-x86-64

SYMBOL TABLE:
no hay símbolos

objdump: la sección '.rodata' se menciona en una opción -j, pero no se encuentra en ningún fichero de entrada
```

3b) ¿Qué dependencias dinámicas tiene su programa? Indique al menos 2 formas de averiguarlo.

La primera forma de ver las dependencias dinámicas del programa es mediante el comando *ldd*, el cual se encarga específicamente de mostrar las librerías compartidas. En la segunda forma se usa el comando *strings*, que extrae cadenas legibles del archivo ejecutable, y *grep*, el cual busca las líneas que contienen referencias a bibliotecas dinámicas, que son las que contienen el sufijo *.so* (shared object).

```
lucas@lucas-Lenovo-V130-15IKB: ~/Escritorio/empotrados/Practicas_SE
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$ ldd programa
linux-vdso.so.1 (0x00007fffaeeb2000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f824408a000)
/lib64/ld-linux-x86-64.so.2 (0x00007f8244299000)
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$ strings programa | grep -i ".so"
/lib64/ld-linux-x86-64.so.2
libc.so.6
```

3c) Con los programas realizados en la tarea 2f) obtenga lo siguiente:

1. Use objdump, u otras órdenes que conozca, para averiguar en qué dirección de la memoria empieza cada función que usted ha definido y cuánto ocupa cada función.

Haciendo uso de los comandos *objdump* y *grep*, se mostrará la dirección de memoria y la cantidad de memoria que ocupan las funciones *SortMatrix* y *suma* para las dos arquitecturas.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ objdump -t mimquina | grep sortMatrix
0000000000000119a g      F .text 0000000000000016a      sortMatrix
(base) cesar@cesar-virtual-machine:~/Desktop$ objdump -t rasp | grep sortMatrix
0000005cf g      F .text 00000013e      sortMatrix
(base) cesar@cesar-virtual-machine:~/Desktop$ objdump -t mimquina | grep suma
00000000000001304 g      F .text 000000000000006c      suma
(base) cesar@cesar-virtual-machine:~/Desktop$ objdump -t rasp | grep suma
00000070d g      F .text 0000005c      suma
```

Como se puede observar, las funciones en el programa con arquitectura ARM ocupan menos espacio.

2. ¿Qué porcentaje de la sección .text ha empleado el compilador para implementar el código que ha definido en C/C++? ¿Qué hay en el resto de la sección .text?

La mayor parte de la sección .text contiene código creado por el compilador, algo que no hemos programado. Para los dos programas se encuentran las mismas funciones, excepto *call_weak_fn* que solo aparece en la arquitectura ARM. Para el programa en ARM también se encuentran una gran cantidad de símbolos creados por el compilador de esta arquitectura.

```
(base) cesar@cesar-virtual-machine:~/Desktop$ objdump -t mimquina | grep .text
00000000000001080 l      d .text 0000000000000000      .text
000000000000010b0 l      F .text 0000000000000000      deregister_tm_clones
000000000000010e0 l      F .text 0000000000000000      register_tm_clones
00000000000001120 l      F .text 0000000000000000      __do_global_ctors_aux
00000000000001160 l      F .text 0000000000000000      frame_dummy
00000000000001490 g      F .text 0000000000000005      __libc_csu_fini
0000000000000119a g      F .text 0000000000000016a      sortMatrix
00000000000001304 g      F .text 000000000000006c      suma
00000000000001420 g      F .text 0000000000000065      __libc_csu_init
00000000000001080 g      F .text 000000000000002f      _start
00000000000001370 g      F .text 00000000000000b0      main
00000000000001169 g      F .text 0000000000000031      swap
(base) cesar@cesar-virtual-machine:~/Desktop$ objdump -t rasp | grep .text
000000494 l      d .text 00000000      .text
000000494 l      d .text 00000000      $t
0000004cc l      d .text 00000000      $d
0000004dc l      d .text 00000000      $a
0000004dc l      F .text 00000000      call_weak_fn
0000004f8 l      d .text 00000000      $d
000000500 l      d .text 00000000      $t
000000501 l      F .text 00000000      deregister_tm_clones
00000051c l      d .text 00000000      $d
00000052c l      d .text 00000000      $t
00000052d l      F .text 00000000      register_tm_clones
000000550 l      d .text 00000000      $d
000000560 l      d .text 00000000      $t
000000561 l      F .text 00000000      __do_global_ctors_aux
00000058c l      d .text 00000000      $d
0000005a0 l      d .text 00000000      $t
0000005a1 l      F .text 00000000      frame_dummy
0000005a4 l      d .text 00000000      $t
000000700 l      d .text 00000000      $d
00000070c l      d .text 00000000      $t
0000007e0 l      d .text 00000000      $d
0000007f4 l      d .text 00000000      $t
00000082c l      d .text 00000000      $d
000000834 l      d .text 00000000      $t
000000835 g      F .text 00000002      __libc_csu_fini
0000005cf g      F .text 00000013e      sortMatrix
00000070d g      F .text 0000005c      suma
0000007f5 g      F .text 00000040      __libc_csu_init
000000495 g      F .text 00000000      _start
000000769 g      F .text 0000008c      main
0000005a5 g      F .text 0000002a      swap
(base) cesar@cesar-virtual-machine:~/Desktop$
```

3. En qué se diferencia el código ensamblador que proporciona gcc con el

parámetro -S y la orden objdump -j .text?

En la figura siguiente se muestra a la izquierda el contenido del archivo en código ensamblador generado y a la derecha el resultado de desensamblar usando *objdump*. En el código generado se observa una representación en ensamblador del código escrito en C. Lo que muestra *objdump*, en cambio, es el resultado del código compilado, por lo que contiene mucha más información acerca del comportamiento del código y sus relaciones con procesos de nivel más bajo.

```

1 .file "tarea2.c"
2 .text
3 .globl A
4 .section .rodata
5 .align 4
6 .type A, @object
7 .size A, 4
8 A:
9 .long 7
10 .text
11 .globl swap
12 .type swap, @function
13 swap:
14 .LFB0:
15 .cfi_startproc
16 endbr64
17 pushq %rbp
18 .cfi_def_cfa_offset 16
19 .cfi_offset 6, -16
20 movq %rsp, %rbp
21 .cfi_def_cfa_register 6
22 movq %rdi, -24(%rbp)
23 movq %rsi, -32(%rbp)
24 movq -24(%rbp), %rax
25 movl (%rax), %eax
26 movl %eax, -4(%rbp)
27 movq -32(%rbp), %rax
28 movl (%rax), %edx
29 movq -24(%rbp), %rax
30 movl %edx, (%rax)
31 movq -32(%rbp), %rax

```

3b) Vamos a preparar dos versiones binarias ARM para su futura ejecución en la tarjeta Raspberry de prácticas. Como no sabemos qué versión de bibliotecas de tiempo de ejecución (runtime libraries) tendremos, generaremos un binario estático y otro dinámico. Investigue cómo generar binarios estáticos con gcc y genere dos versiones binarias estáticas para ARM y x86-64. ¿Qué salida obtiene al ejecutar ldd sobre los ejecutables para x86_64?

Para crear binarios estáticos en gcc se usa el argumento `-static`. El comando `ldd` imprime las dependencias compartidas de un archivo ejecutable. Como se trata de un archivo estático, no tiene bibliotecas compartidas por lo que salta el error que se muestra.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ gcc tarea2.c -o tarea2_x86-64 -static
tarea2.c: In function 'main':
tarea2.c:55:10: warning: type defaults to 'int' in declaration of 'ELF_auto_sumaContenido' [-Wimplicit-int]
   55 |         auto ELF_auto_sumaContenido = 0;
      |         ^
      |         ^~~~~~
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ ldd tarea2_x86-64
not a dynamic executable
```

Para crear el archivo ARM se usará el compilador *arm-linux-gnueabi-hf-gcc*. Al también ser estático, ocurre lo mismo.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ arm-linux-gnueabi-gcc tarea2.c -o tarea2_ARM_static -static
tarea2.c: In function 'main':
tarea2.c:55:10: warning: type defaults to 'int' in declaration of 'Elf_auto_sumaContenido' [-Wimplicit-int]
   55 |     auto Elf_auto_sumaContenido = 0;
      |          ^
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ ldd tarea2_ARM_static
not a dynamic executable
```

Al crear el archivo x86-64 sin usar el argumento *-static*, se crea con enlace dinámico por lo que esta vez al usar el comando *ldd* se muestran las bibliotecas compartidas a las que está enlazado.


```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ gcc tarea2.c -o tarea2
tarea2.c: In function 'main':
tarea2.c:55:10: warning: type defaults to 'int' in declaration of 'ELF_auto_sumaContenido' [-Wimplicit-int]
   55 |     auto ELF_auto_sumaContenido = 0;
      |         ^
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ ldd tarea2
linux-vdso.so.1 (0x00007ffeb49c5000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f93d1c00000)
/lib64/ld-linux-x86-64.so.2 (0x00007f93d1fea000)
```

3c) Suponga que queremos entregar una versión hexadecimal de nuestro programa ARM. Estas versiones se usan para cargar directamente el programa y los datos en las direcciones de memoria que se indican, y NO dependen de qué compilador las ha producido. Encuentre la forma de hacerlo con la orden `objcopy`.

El comando `objcopy` permite, entre otras acciones, la conversión a diferentes formatos. En este caso se le indica por parámetros que se debe transformar un binario en un archivo Intel HEX.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ objcopy -I binary -O ihex tarea2_ARM tarea2_ARM_hex.hex
```

TAREA 4

Ejecute con QEMU de forma standalone/SO-less algunos de los programas de la primera parte de la práctica para ARM. ¿Observa algún error? Explíquelo e investigue a qué es debido.

En la siguiente imagen se puede observar que al intentar ejecutar el archivo compilado de forma cruzada para un sistema ARM salta el fallo *no such file or directory*. Utilizando los dos métodos propuestos en el guión se comprueba que, efectivamente, en el sistema utilizado no se encuentra la librería que provoca el fallo ni otra adicional. En caso de solucionar únicamente el primer fallo, probablemente saltaría otro similar con la segunda biblioteca ausente.

```
lucas@lucas-Lenovo-V130-15IKB: ~/Escritorio/empotrados/Practicas_SE
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$ qemu-arm rasp
/lib/ld-linux-armhf.so.3: No such file or directory
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$ arm-none-eabi-readelf -a rasp |grep -i "Shared library"
0x00000001 (NEEDED)             Shared library: [libc.so.6]
0x00000001 (NEEDED)             Shared library: [ld-linux-armhf.so.3]
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$ objdump -x rasp |grep -i "NEEDED"
NEEDED               libc.so.6
NEEDED               ld-linux-armhf.so.3
(base) lucas@lucas-Lenovo-V130-15IKB:~/Escritorio/empotrados/Practicas_SE$
```

TAREA 5

Junto a este guion encontrara los siguientes 4 programas: `prog_a.elf`, `prog_b.elf`, `prog_c.elf` y `prog_d.elf`. La ejecución de dichos programas esconde un mensaje secreto ;-). Debe ejecutar estos programas para descubrir el mensaje.

En primer lugar se usará el comando `file` para averiguar la arquitectura para la que fueron compilados los programas.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ file prog_a.elf
prog_a.elf: ELF 64-bit LSB pie executable, UCB RISC-V, RVC, double-float ABI, version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux-riscv64-lp64d.so.1, BuildID[sha1]=771ac84754613ec661d2e24447d602ae3935cde1, for GNU/Linux 4.15.0, not stripped
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ file prog_b.elf
prog_b.elf: ELF 32-bit LSB pie executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.3, BuildID[sha1]=72dd67d9faa58e7fc69d0f20ce9ae83f2d743c56, for GNU/Linux 3.2.0, not stripped
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ file prog_c.elf
prog_c.elf: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), statically linked, BuildID[sha1]=3626d7d6aa35330ec44147e212dc19f12e48, for GNU/Linux 3.2.0, not stripped
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ file prog_d.elf
prog_d.elf: ELF 64-bit LSB executable, UCB RISC-V, RVC, double-float ABI, version 1 (SYSV), statically linked, BuildID[sha1]=128237a12d81c13adff329279da236a628f1c64c, for GNU/Linux 4.15.0, not stripped
```


Las características más relevantes del ISA para el que fueron compilados se muestran a continuación.

prog_a.elf: 64-bit, arquitectura RISC-V, enlace dinámico.

prog_b.elf: 32-bit, arquitectura ARM, enlace dinámico.

prog_c.elf: 32-bit, arquitectura ARM, enlace estático.

prog_d.elf: 64-bit, arquitectura RISC-V, enlace estático.

Los mensajes de los programas con enlace estático se pueden mostrar simplemente ejecutándolos con su arquitectura usando qemu.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ qemu-arm prog_c.elf
Si el mundo es un pañuelo, ¡nosotros qué somos!
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ qemu-riscv64 prog_d.elf
Qué le dice una impresora a otra? Esta hoja es tuYa o es impresión mía.
```

Sin embargo, al tratar de ejecutar los programas enlazados dinámicamente salta error.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ qemu-riscv64 prog_a.elf
qemu-riscv64: Could not open '/lib/ld-linux-riscv64-lp64d.so.1': No such file or directory
```

Por lo que para que el programa pueda acceder a la biblioteca solicitada primero habrá que instalarla.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ sudo apt install gcc-riscv64-linux-gnu
Reading package lists... Done
Building dependency tree... Done
Reading state information... Done
The following additional packages will be installed:
```

Luego se comprueba que efectivamente se dispone del enlazador.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ ls /usr/riscv64-linux-gnu/lib
crt1.o          libatomic.so.1.2.0  libdl.so.2        libnss_compat.so   libresolv.so.2
crti.o          libBrokenLocale.a   libg.a            libnss_compat.so.2 librt.a
crtm.o          libBrokenLocale.so  libgcc_s.so.1     libnss_dns.so.2    librt.so.1
gcrt1.o         libBrokenLocale.so.1 libgomp.so.1      libnss_files.so.2  libthread_db.so
ld-linux-riscv64-lp64d.so.1 libc.a              libgomp.so.1.0.0  libnss_hesiod.so   libthread_db.so.1
libanl.a        libc_malloc_debug.so libm.a            libnss_hesiod.so.2 libutil.a
libanl.so       libc_malloc_debug.so.0 libmcheck.a       libpcprofile.so    libutil.so.1
libanl.so.1     libc_nonshared.a    libmemusage.so    libpthread.a       M crt1.o
```

Y finalmente se enlaza la ruta de la biblioteca compartida con el archivo para que pueda usarla.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ qemu-riscv64 -L /usr/riscv64-linux-gnu/ prog_a.elf
La Alhambra es la mejor cerveza del mundo.zCruzcampo? Suspenso!
```

Lo mismo ocurre en el caso del programa b, que necesita *ld-linux.so.3*.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ sudo apt install gcc-arm-linux-gnueabi
```

Y al enlazar ahora sí se podrá ejecutar los programas y ver correctamente el mensaje.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ qemu-arm -L /usr/arm-linux-gnueabi prog_b.elf
Non si vede bene che col cuore. L'essenziale è invisibile agli occhi
```

TAREA 6

6.1) ¿Cuál es el ISA para el que ha sido definido prog_e.elf?

Procediendo de forma similar a la tarea anterior obtenemos que la arquitectura para la que este programa fue compilado es ARM de 32 bits y que está enlazado dinámicamente.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ file prog_e.elf
prog_e.elf: ELF 32-bit LSB pie executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.3, BuildID[sha1]=497f91d95bd2f864963bafdb3a4dbad4990b7ab, for GNU/Linux 3.2.0, not stripped
```

6.2) En las posiciones hexadecimales 0x11048 y 0x17e98 de la sección de datos inicializados se encuentran dos variables que ocupan 28234 y 81456 bytes respectivamente. ¡Se tratan de blobs secretos escondidos en el ejecutable!

La sección de datos inicializados es la `.data`. Se pueden ver los archivos pertenecientes a esta sección, donde se encuentran los dos blobs.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ objdump -t prog_e.elf | grep '.data'
000006b0 l d .rodata 00000000 .rodata
00011040 l d .data 00000000 .data
000006b0 l .rodata 00000000 $d
00011040 l .data 00000000 $d
000006b4 l .rodata 00000000 $d
000006b4 l 0 .rodata 00000000 all IMPLIED_fbits
00011044 l .data 00000000 $d
00011048 l .data 00000000 $d
0002bcd0 l 0 .data 00000049 cad
00000744 l .rodata 00000000 $d
00000744 l 0 .rodata 00000000 all IMPLIED_fbits
00011040 w .data 00000000 data_start
0002bccc g 0 .data 00000004 _FEO_
0002bd24 g .data 00000000 _edata
00011048 g 0 .data 00006e4a blob1
0002bd1c g 0 .data 00000004 _i_
00017e94 g 0 .data 00000004 blob1_len
00011040 g .data 00000000 _data_start
0002bcc8 g 0 .data 00000004 blob2_len
00011044 g 0 .data 00000000 .hidden __dso_handle
000006b0 g 0 .rodata 00000004 .hidden __IO_stdin_used
0002bd20 g 0 .data 00000004 _j_
00017e98 g 0 .data 00013e30 blob2
0002bd24 g 0 .data 00000000 .hidden __TMC_END__
```

Si se utiliza el comando `objdump` y acotamos con `grep` se puede ver la dirección virtual en la que comienza la sección `.data` en la memoria del programa. En este caso es 0x11040. También muestra el offset en el archivo ELF en el que comienza la sección, el cual es de 0x1040.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ objdump -h prog_e.elf | grep '.data'
14 .rodata 00000124 000006b0 000006b0 000006b0 2**2
21 .data 0001ace4 00011040 00011040 00011040 2**2
```

Para obtener la dirección física donde se encuentran los archivos, se deberá tener en cuenta el offset y la dirección virtual de la sección:

$$\text{Dirección física blob} = (\text{dirección blob en sección} - \text{dirección comienzo sección}) + \text{offset sección}$$

Se aplicará esta fórmula a ambos blobs:

- **Dirección Blob 1:** $(0x11048 - 0x11040) + 0x1040 = 0x1048$ (4168 en decimal)
- **Dirección Blob 2:** $(0x17e98 - 0x11040) + 0x1040 = 0x7e98$ (32408 en decimal)

Conocidas las direcciones físicas de los blobs, se volcarán mediante el comando `dd`, el cual sirve para copiar bloques de datos. Como argumentos se le pasará el archivo del que se extraerán los datos (input file), el nombre del archivo donde se almacenarán (output file), del número de bytes

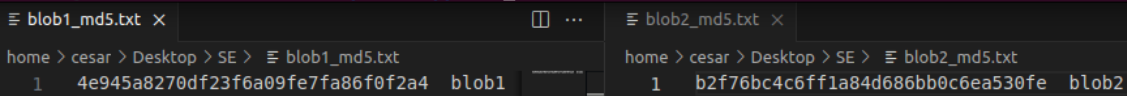
que leerá por paso (input block size), los bytes que saltará hasta empezar a copiar (skip) y el número de bytes que copiará (count).

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ dd if=prog_e.elf of=blob1 ibs=1 skip=4168 count=28234
28234+0 records in
55+1 records out
28234 bytes (28 kB, 28 KiB) copied, 0,00792147 s, 3,6 MB/s
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ dd if=prog_e.elf of=blob2 ibs=1 skip=32408 count=81456
81456+0 records in
159+1 records out
81456 bytes (81 kB, 80 KiB) copied, 0,0300786 s, 2,7 MB/s
```

Una vez extraídos los archivos se podrá contestar a los apartados de la sección 6.2:

- Para obtener su valor MD5 se usará el comando *md5sum* y se volcará en un fichero de texto.

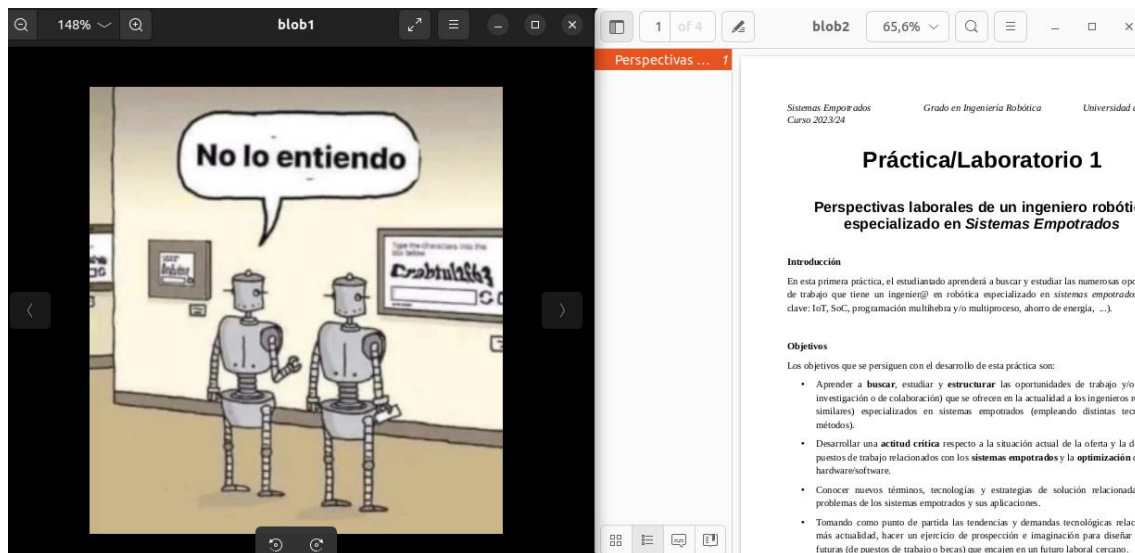
```
cesar@cesar-virtual-machine:~/Desktop/SE$ md5sum blob1 > blob1_md5.txt
cesar@cesar-virtual-machine:~/Desktop/SE$ md5sum blob2 > blob2_md5.txt
cesar@cesar-virtual-machine:~/Desktop/SE$
```



- El formato se obtendrá mediante el comando *file*.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ file blob1
blob1: JPEG image data, JFIF standard 1.01, resolution (DPI), density 72x72, segment length 16, Exif Standard: [
TIFF image data, little-endian, direntries=7, orientation=upper-left, xresolution=98, yresolution=106, resolutio
nunit=2, software=GIMP 2.10.34, datetime=2023:09:24 19:37:39], progressive, precision 8, 302x326, components 3
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ file blob2
blob2: PDF document, version 1.6, 1 pages
```

Parece indicar que el blob1 se trata de una imagen jpeg y el blob2 de un archivo pdf. Efectivamente, si abrimos los archivos se puede comprobar que el primer blob muestra un meme robótico mientras que el segundo se corresponde con el guión de la primera práctica del curso.



Además, ejecutando el programa de forma similar al ejercicio 5, se encuentra un nuevo mensaje secreto.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ qemu-arm -L /usr/arm-linux-gnueabi prog_e.elf
Si los Zombies se deshacen con el paso del tiempo ... Zombiodegradables?
```

6.3) En la posición 0x2bd20 de la sección de datos inicializados se encuentra, codificado en hexadecimal, lo que hace Paquito todas las mañanas. ¿De qué se

trata? Tenga en cuenta el **ENDIANESS** de la arquitectura y que se trata de un entero de 32 bits.

De forma similar al apartado anterior, se calculará la dirección de interés y se extraerá la información solicitada.

- **Paquito:** $(0x2bd20 - 0x11040) + 0x1040 = 0x1bd20$ (113952 en decimal)

Como se trata de un entero de 32 bits (4 bytes), el parámetro *count* valdrá 4. Después se usará el comando *xxd* para la visualización de la información en formato binario y hexadecimal.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ dd if=prog_e.elf of=paquito ibs=1 skip=113952 count=4
4+0 records in
0+1 records out
4 bytes copied, 0,000197969 s, 20,2 kB/s
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ xxd paquito
00000000: caca 0000                                ....
```

Se puede observar que lo que hace Paquito todas las mañanas es caca.

6.4) En la posición 0x2bd1c de la sección de datos inicializados se encuentra, codificado en hexadecimal, lo que se toma Peter todas las mañanas en bocadillo. ¿De qué se trata? Tenga en cuenta el ENDIANESS de la arquitectura y que se trata de un entero de 64 bits.

Se procederá de idéntica forma al apartado anterior.

- **Peter:** $(0x2bd1c - 0x11040) + 0x1040 = 0x1bd1c$ (113948 en decimal)

En este caso se trata de un entero de 64 bits (8 bytes) por lo que *count* valdrá 8.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ dd if=prog_e.elf of=peter ibs=1 skip=113948 count=8
8+0 records in
0+1 records out
8 bytes copied, 0,000334534 s, 23,9 kB/s
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ xxd peter
00000000: afbe adde caca 0000                    .....
```

En el anterior no influía, pero en este apartado se deberá prestar atención al endianness para descifrar el mensaje. Mediante el comando *file* se puede observar que *prog_e.elf* usa un formato LBS, el cual significa “Least Significant Byte”. Esto significa que el archivo es little-endian, por lo que se puede deducir que lo que se toma Peter todas las mañanas es beaf.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ file prog_e.elf
prog_e.elf: ELF 32-bit LSB pie executable, ARM, EABI5 version 1 (SYSV), dynamically linked, interpreter /lib/ld-linux.so.3, BuildID[sha1]=497f91d95bd2f864963bafdb3a4dbad4990b7ab, for GNU/Linux 3.2.0, not stripped
```

6.5) ¿Cuál es la segunda instrucción ensamblador de la función main? ¿Cómo se está codificada en hexadecimal?

Primero se desensamblará el archivo.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ aarch64-linux-gnu-objdump -j .text -d prog_e.elf
prog_e.elf:      file format elf32-littlearm

Disassembly of section .text:

00000420 <_start>:
420:  e3a0b000      mov     fp, #0
```

Una vez encontrada la función *main*, se puede ver su código ensamblador. La representación hexadecimal de la segunda instrucción, la cual se encuentra en la dirección de memoria 0x5a0, es e28db004.

```
0000059c <main>:
59c: e92d4800    push    {fp, lr}
5a0: e28db004    add     fp, sp, #4
5a4: e24dd008    sub     sp, sp, #8
5a8: e3a03000    mov     r3, #0
5ac: e50b3008    str     r3, [fp, #-8]
5b0: ea00000f    b       5f4 <main+0x58>
5b4: e59f2078    ldr     r2, [pc, #120] ; 634 <main+0x98>
5b8: e08f2002    add     r2, pc, r2
5bc: e51b3008    ldr     r3, [fp, #-8]
5c0: e0823003    add     r3, r2, r3
5c4: e5d33000    ldrb    r3, [r3]
```

TAREA 7

En esta tarea vamos a ejecutar/depurar el programa de forma cruzada empleando QEMU y la versión ARM de gdb que tiene instalada en el laboratorio de prácticas.

Para esta tarea se usará el archivo `tarea2_ARM_static`, el cual tiene arquitectura ARM de 32 bits. A continuación, se llamará a QEMU en modo stand-alone y se ejecutará el programa por un puerto TCP.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ qemu-arm -g 1112 tarea2_ARM_static
```

Luego abrirá otra terminal y se entrará en gdb para controlar remotamente la ejecución de este programa.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ gdb-multiarch tarea2_ARM_static
GNU gdb (Ubuntu 12.1-0ubuntu1~22.04) 12.1
Copyright (C) 2022 Free Software Foundation, Inc.
License: GPL+3; GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
```

Después, se conectará con el proceso QEMU que espera en la otra terminal usando el mismo puerto TCP:

```
(gdb) target remote localhost:1112
Remote debugging using localhost:1112
0x00010338 in _start ()
(gdb)
```

¿Qué salida observa? ¿Dónde está parada la ejecución a la espera de sus órdenes desde GDB? ¿Por qué no se para en la función *main*, que es la primera que escribió en su programa C/C++?

Al conectar con el proceso QEMU desde gdb, se observa que el depurador se para en el proceso `_start`. Este realiza tareas fundamentales de bajo nivel como la inicialización de registros o la configuración de la pila, por lo que este es el inicio real del programa y hasta que no se haya ejecutado este proceso no se invocará la función *main*.

Proceda a la ejecución paso del programa.

Al continuar con la ejecución paso a paso mediante la ejecución remota, se consigue completar el programa y obtener el resultado.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ qemu-arm -g 1112 tarea2_ARM_static 45
```

Pruebe a generar los binarios con información para depuración. Investigue cómo hacerlo. Compruebe también cómo se generan más secciones de código y datos con `objdump -h` y que los programas ocupan más espacio.

Para generar binarios con información para depuración, se usará el parámetro `-g3` de `gcc`.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ gcc -g3 -o tarea2_bin_debug tarea2.c
```

Al comprobar el número de secciones de código creadas con un ejecutable básico,

```
25 .bss          00000008 00000000000004010 00000000000004010 00003010 2**0
                ALLOC
26 .comment      0000002b 00000000000000000 00000000000000000 00003010 2**0
                CONTENTS, READONLY
```

en comparación con el que se acaba de crear que incluye información para depuración,

```
32 .debug_line_str 00000222 00000000000000000 00000000000000000 000092da 2**0
                CONTENTS, READONLY, DEBUGGING, OCTETS
33 .debug_macro   00001514 00000000000000000 00000000000000000 000094fc 2**0
                CONTENTS, READONLY, DEBUGGING, OCTETS
```

observa que este último archivo tiene más secciones. Además, si se obtiene la información sobre cuanto ocupa cada programa, se puede ver que este último binario ocupa casi tres veces más que el base.

```
(base) cesar@cesar-virtual-machine:~/Desktop/SE$ ls -l tarea2_bin_debug tarea2
-rwxrwxr-x 1 cesar cesar 16136 oct 20 14:03 tarea2
-rwxrwxr-x 1 cesar cesar 47856 oct 26 11:33 tarea2_bin_debug
```

CUADERNO DE BITÁCORA

- 25 SEPTIEMBRE -

En la primera sesión de la práctica 2, el grupo dividió equitativamente los apartados de la tarea 1 y se acordó llevar hechos dichos ejercicios a la siguiente sesión para poder seguir avanzando con la práctica al disponer de los conceptos básicos.

- 2 OCTUBRE -

Se han traído resueltos los ejercicios de la tarea 1, salvo el apartado b. En clase se han elaborado los ejemplos que pide la práctica utilizando diferentes comandos de la terminal como *grep*, *printf* o *wc*. Además, se ha comenzado la tarea 2 y nos hemos distribuido el trabajo de cara a las siguientes sesiones.

- 9 OCTUBRE -

Se ha avanzado en el desarrollo del ejercicio 2, completando los dos primeros apartados. Debido a la distribución de tareas, se ha comenzado la tarea 3, de la cual se han completado los apartados a y b. Se ha puesto en marcha un repositorio de Github para sincronizar los archivos entre los dos sistemas que se utilizan.

- 16 OCTUBRE -

Se ha completado el ejercicio 2, salvo el apartado f, y se han avanzado los ejercicios 3, 4 y 5. Además, se ha hecho una distribución final acorde a la tarea restante.

- 23 OCTUBRE -

Se han traído hechos los ejercicios que faltaban, según la distribución final de la semana anterior. Para hacer en conjunto en clase se dejó el apartado 2f y el ejercicio 7, los cuales se han completado hoy.