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.

A screenshot of a computer

Description automatically generated

* **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.

A close up of a text

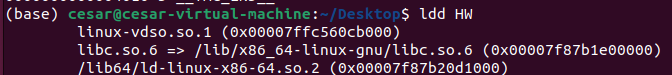
Description automatically generated

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

A screenshot of a computer program

Description automatically generated

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



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

A computer screen shot of a computer

Description automatically generated

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

A screen shot of a computer screen

Description automatically generated

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

A black background with white text

Description automatically generated

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

A screen shot of a computer program

Description automatically generated

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

A screenshot of a computer

Description automatically generated

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

A computer screen shot of white text

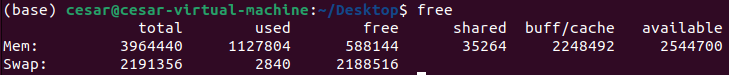
Description automatically generated

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

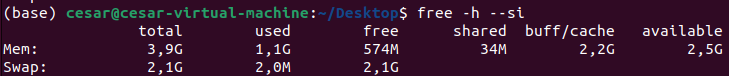
A screenshot of a computer

Description automatically generated

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



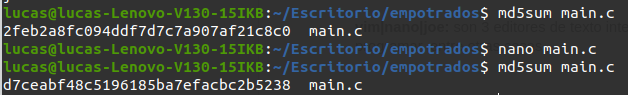
Usando los argumentos adicionales -h y --si, se obtiene una lectura más comprensible.



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



* **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.



* **vim|nano|joe:** son 3 editores de texto integrados en el entorno de la terminal, sirven para programar, entre otras cosas.
* Vim: A black screen with text

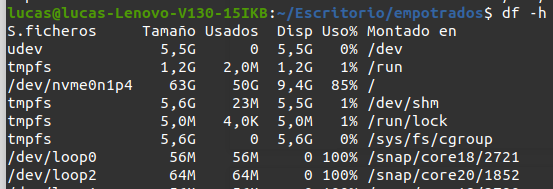
  Description automatically generated
* Nano:A screenshot of a computer

  Description automatically generated
* Joe: A screenshot of a computer

  Description automatically generated
* **sudo:** otorga privilegios de administrador al usuario para la ejecución de ese comando.
* **ld:** 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.



* **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.



**1b) Incluya 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 sóla 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.

A screenshot of a computer program

Description automatically generated

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.

A screenshot of a computer

Description automatically generated

**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 inicializan 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 inicializan 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 ejecucion de un programa, explique la diferencia entre la pila (stack) y el monticulo (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 auomática 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) // 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);

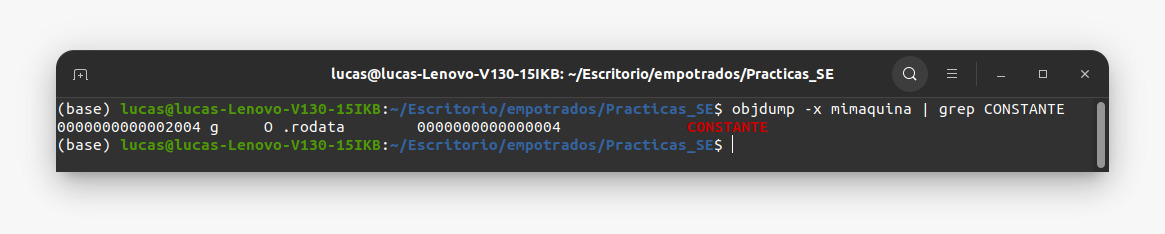
printf("%d \n",ELF\_auto\_sumaContenido);

return 0;

}

2b) 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.

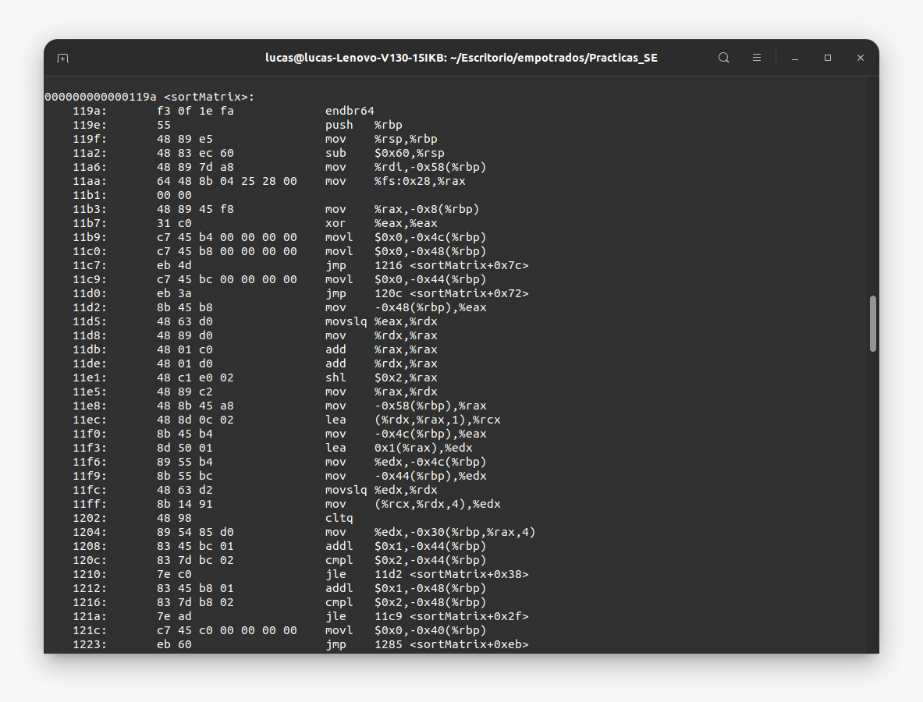
2c) Con la siguiente concatenación de comandos se pueden visualizar las variables estáticas del programa:

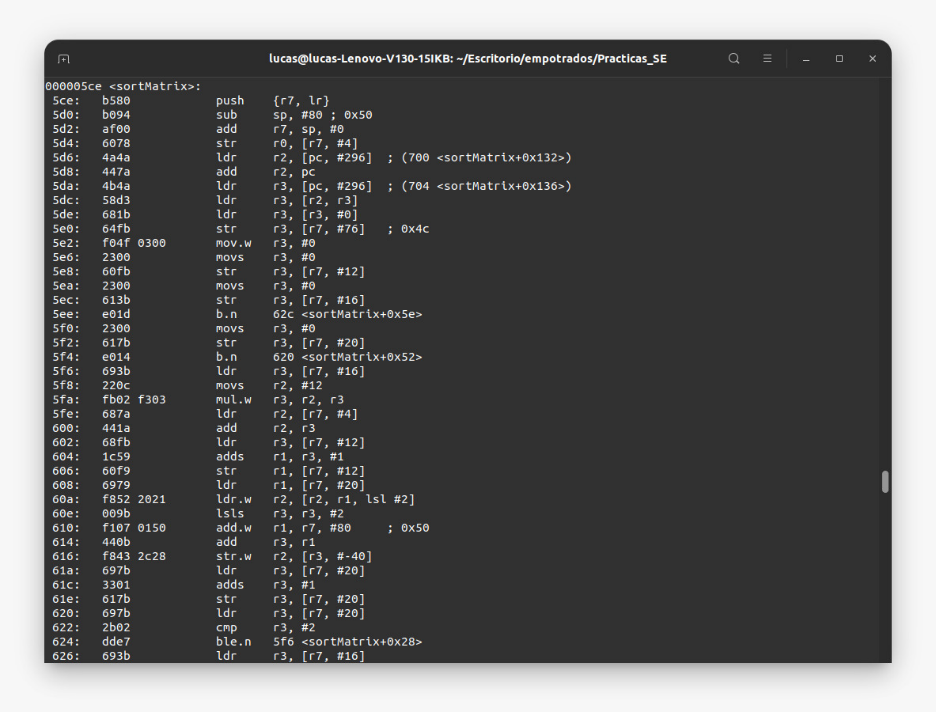


2d) La diferencia reside en su ciclo de vida. Una variable estática dura tanto como dure la ejecución del programa y mantiene su valor. Una variable dinámica se crea en tiempo de ejecución y su ciclo de vida es controlado por el programa en cuestión, pudiendo eliminarse antes de que termine. Finalmente, 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) 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.Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

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.



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.

A computer screen shot of a computer program

Description automatically generated

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

A screen shot of a computer program

Description automatically generated

**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 *string*, 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).

**A screen shot of a computer program

Description automatically generated**

**3c) Con los programas realizados en la tarea 2f) (para las dos arquitecturas) 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.**

**-------------------------------------------------------------------------------------------------------------------------------**

**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* imprimie 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.

A computer screen shot of text

Description automatically generated

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

A computer screen shot of text

Description automatically generated

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.

A computer screen shot of a computer code

Description automatically generated

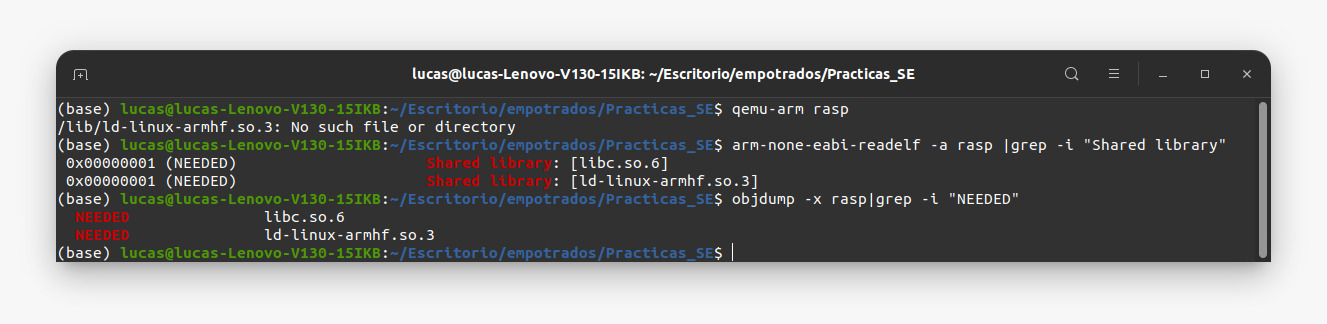
**3c) Suponga que queremos entregar una versión hexadecimal de nuestro programa ARM. Estas versiones se usan para cargar directamente el programa y los datos en las direcciones de memoria que se indican, y NO dependen de qué compilador las ha producido. Encuentre la forma de hacerlo con la orden objcopy.**

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.



tarea 4

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.



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 ejecucion 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.

A computer screen shot of a program

Description automatically generated

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

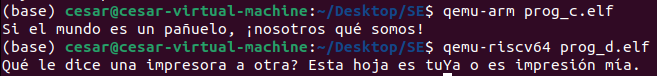
**prog\_a.elf**: 64-bit, arquitectura RISC-V, enlace dinámico.

**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.

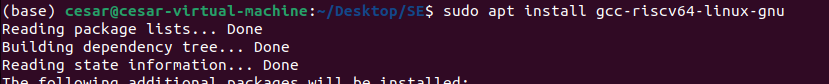
Los mensajes de los programas con enlace estático se pueden mostrar simplemente ejecutándlos con su arquitectura usando qemu.



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



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



Luego se comprueba que efectivamente se dispone del enlazador.

A computer screen shot of a program

Description automatically generated

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



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



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



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.



**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.

A screenshot of a computer program

Description automatically generated

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.



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

*Dirección física blob = (dirección blob en sección – dirección comienzo sección) + 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).

A computer screen with text

Description automatically generated

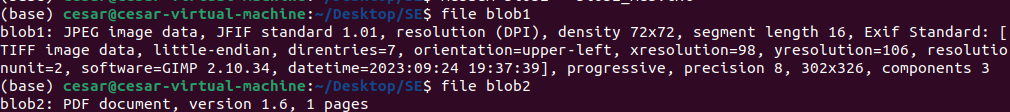
Una vez extraídos los archivos se podrá contestar a los apartados de la sección 6.2:

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

A screenshot of a computer

Description automatically generated

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



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.

A screenshot of a computer

Description automatically generated

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

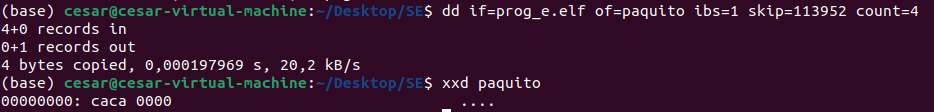


**6.3) En la posición 0x2bd20 de la sección de datos inicializados se encuentra, codificado en hexadecimal, lo que hace Paquito todas las mañanas. ¿De qué se trata? Tenga en cuenta el ENDIANESS de la arquitectura y que se trata de un entero de 32 bits.**

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.



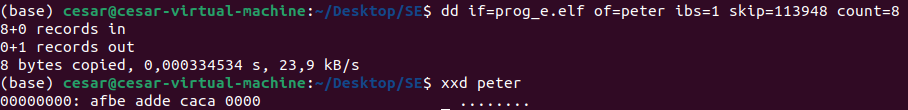
Se puede observar que lo que hace Paquito todas las mañanas es caca.

**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.



En el anterior no influía, pero en este apartado se deberá prestar atención al endianess 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.



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

Primero se desensamblará el archivo.

A computer screen with text

Description automatically generated

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.

A computer screen shot of white text

Description automatically generated

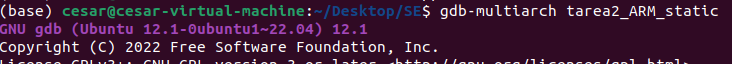
tarea 7

**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.



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



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

A screenshot of a computer screen

Description automatically generated

**¿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.



**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.



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

A computer screen shot of numbers

Description automatically generated

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

A computer screen with white text

Description automatically generated

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.

