Universidad de San Andrés

I301 - Arquitectura de computadoras y Sistemas Operativos

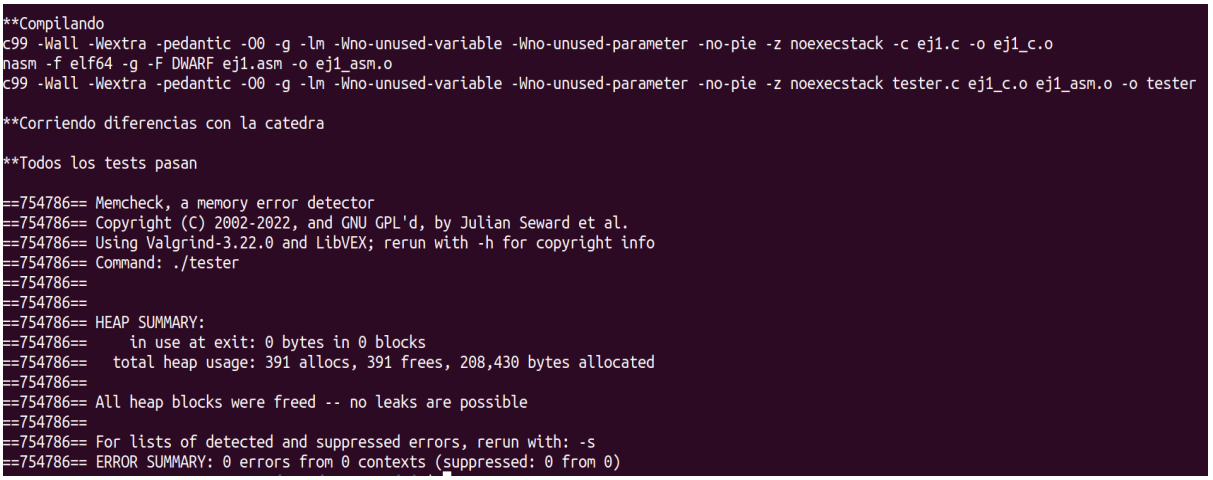
Fecha de entrega: **16/04/2025**

Criterio de aprobación

Este trabajo práctico se compone de dos ejercicios, en el primer ejercicio se deben implementar unas funciones específicas utilizando el lenguaje de programación C y Assembler. El segundo ejercicio consta de analizar un ejecutable **bomb** y descifrar, con las herramientas que se provee en el enunciado, cuáles son los inputs que necesita para que esta no *explote*.

Criterio de aprobación del ejercicio 1:

Tiene que pasar **todos** los test (**runTester.sh** tiene que dar ok, como la captura) que se proveen con el ejercicio tanto para la implementación en C y Assembly.



```
**Compilando
c99 -Wall -Wextra -pedantic -O0 -g -lm -Wno-unused-variable -Wno-unused-parameter -no-pie -z noexecstack -c ej1.c -o ej1_c.o
nasm -f elf64 -g -F DWARF ej1.asm -o ej1_asm.o
c99 -Wall -Wextra -pedantic -O0 -g -lm -Wno-unused-variable -Wno-unused-parameter -no-pie -z noexecstack tester.c ej1_c.o ej1_asm.o -o tester

**Corriendo diferencias con la catedra

**Todos los tests pasan

==754786== Memcheck, a memory error detector
==754786== Copyright (C) 2002-2022, and GNU GPL'd, by Julian Seward et al.
==754786== Using Valgrind-3.22.0 and LibVEX; rerun with -h for copyright info
==754786== Command: ./tester
==754786==
==754786==
==754786== HEAP SUMMARY:
==754786==     in use at exit: 0 bytes in 0 blocks
==754786==   total heap usage: 391 allocs, 391 frees, 208,430 bytes allocated
==754786==
==754786== All heap blocks were freed -- no leaks are possible
==754786==
==754786== For lists of detected and suppressed errors, rerun with: -s
==754786== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Criterio de aprobación del ejercicio 2:

En este caso se considera aprobado aquellas entregas que lleguen a desactivar hasta las fase 3 (inclusive) de la bomba.

Ejercicio 1

El objetivo de este ejercicio es tener una primera experiencia programando en assembler y además solidificar los conceptos sobre la pila que necesitaran para el próximo ejercicio.

Antes de empezar:

En el directorio de este ejercicio encontrarán los siguientes archivos:

- Makefile
- ej1.asm
- ej1.c
- ej1.h
- main.c
- runMain.sh
- runTester.sh
- tester.c
- salida.catedra.ej1.txt

Los archivos sobre los que deben trabajar son “ej1.asm” y “ej1.c”, aquí deberán implementar las funciones solicitadas en este enunciado.

El archivo "ej1.h" contiene las declaraciones de las estructuras y funciones. En la línea 8 de este archivo se encuentra la variable `USE_ASM_IMPL`, la cual está inicializada en 1 para probar la implementación en ensamblador. Si se desea evaluar la implementación en C, es necesario cambiar su valor a 0.

El archivo “main.c” es para que ustedes realicen sus propias pruebas. Siéntanse a gusto de manejarlo como crean conveniente. Para compilar el código y poder correr las pruebas cortas implementadas en main, deberá ejecutar:

$ make main
$ ./runMain.sh

“runMain.sh” verifica que no se pierde memoria ni se realizan accesos incorrectos a la misma.

Para compilar el código y correr las pruebas intensivas, deberá ejecutar:

$ ./runTester.sh

En este punto se usa el archivo **salida.catedra.ej1.txt** para comparar, la salida que produce su código con el de la cátedra. En caso de que haya discrepancias pueden usar el archivo **salida.caso.propio.ej1.txt** (que se autogenera) para comparar en qué punto concreto se necesita el fix.

El objetivo específico de este ejercicio es implementar un conjunto de funciones sobre una lista doblemente enlazada de tamaño variable. **Las funciones deberán ser implementadas en C y en Assembler**

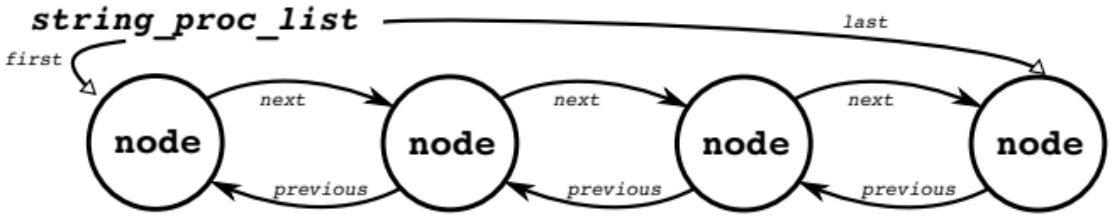


Figura 1: Ejemplo de Lista Doblemente Enlazada.

Para ello, tenemos una estructura que indica el primer y el último nodo, y cada nodo tiene punteros al anterior y al siguiente. Además, cada nodo tiene un tipo y un hash. Definición de las estructuras:

```
/** Lista **/
typedef struct string_proc_list_t {
    struct string_proc_node_t* first;
    struct string_proc_node_t* last;
} string_proc_list;

/** Nodo **/
typedef struct string_proc_node_t {
    struct string_proc_node_t* next;
    struct string_proc_node_t* previous;
    uint8_t type;
    char* hash;
} string_proc_node;
```

Funciones a implementar:

1

```
string_proc_list* string_proc_list_create(void);
string_proc_list* string_proc_list_create_asm(void);
```

Inicializa una estructura de lista.

2

```
string_proc_node* string_proc_node_create(uint8_t type, char* hash);
string_proc_node* string_proc_node_create_asm(uint8_t type, char* hash);
```

Inicializa un nodo con el tipo y el hash dado.
El nodo tiene que apuntar al hash pasado por parámetro (no hay que copiarlo).

3

```
void string_proc_list_add_node(string_proc_list* list, uint8_t type, char* hash);
void string_proc_list_add_node_asm(string_proc_list* list, uint8_t type, char* hash);
```

Agrega un nodo nuevo al final de la lista con el tipo y el hash dado.

Recordar que no se debe copiar el hash, sino apuntar al mismo.

4

```
char* string_proc_list_concat(string_proc_list* list, uint8_t type, char* hash);
char* string_proc_list_concat_asm(string_proc_list* list, uint8_t type, char* hash);
```

Genera un nuevo hash concatenando el pasado por parámetro con todos los hashes de los nodos de la lista cuyos tipos coinciden con el pasado por parámetro

Ejercicio 2

Este trabajo se basa en un ejercicio desarrollado en la Universidad Carnegie Mellon por los profesores R. Bryant y D. O'Hallaron.

Hay una única bomba por alumno están indexadas de acuerdo a cómo se anotaron en la planilla de entregas:

Entregas TPs - Hojas de cálculo de Google

Las bombas están en el server en la ruta: **/network/publico.** Copien a su home en el servidor la que corresponda según **Bomb #.** Una vez copiada la bomba, en la planilla completen con **OK** la columna **descargada.** Es muy importante que verifiquen que están copiando la bomba correcta de acuerdo con la planilla, ya que la corrección se realiza de forma automática y no evaluará bombas que no correspondan a la asignada**.**

Asegúrense de que los siguientes archivos estén en su directorio, ya que la bomba los utiliza todos:

a. **ID : identificador único por alumno.**
b. **bomb: código objeto compilado.**
c. **palabras.txt: Diccionario de palabras.**
d. **bomb.c: ejecuta cada una de las fases de la bomba.**
e. **gdb_refcard_gnu.pdf**

Fuera del servidor.

Lo siguiente es por si prefieren trabajar local en lugar del servidor:

En su carpeta hay un archivo llamado **`.gdbinit`**. El punto al inicio lo convierte en un archivo oculto en Linux. Para verlo, pueden ejecutar el siguiente comando:

```
$ls -alh   # la a es all
```

Muevan este archivo a su home, para tener todas las configuraciones correctas de **gdb**, con el siguiente comando:

```
$mv $dir_con_el_archivo/.gdbinit ~/
```

2. En el caso de que deseen ver la próxima instrucción de assembly a ejecutarse, con gdb corriendo pueden usar los siguientes flags:

```
$gdb bomb
(gdb) set disassemble-next-line on
(gdb) show disassemble-next-line
```

1 - Un poco de contexto:

En un mundo donde la tecnología domina cada aspecto de nuestra vida, un grupo de estudiantes de informática se enfrenta a un desafío sin precedentes: han sido seleccionados para participar en una misión crítica. Su objetivo es desactivar un código malicioso conocido como ***'bombas binarias***', diseminado en la red por un grupo de hackers. Si no logran desactivarlas a tiempo, estas bombas explotarán vulnerabilidades en los sistemas, permitiendo el robo masivo de datos de usuarios. La misión es liberar las claves que desactivan las bombas antes de que sea demasiado tarde.

Sin el código fuente original, no tenemos mucho por dónde empezar, pero hemos observado que los programas parecen operar en una secuencia de niveles o fases. ***Hay 4 fases en total***. Cada nivel desafía al usuario a ***ingresar una cadena de texto, numérica o ambas al mismo tiempo. La bomba también permite que se le provea un archivo de texto con las claves correspondientes***. Si el usuario ingresa la cadena correcta, desactiva el nivel, el código malicioso es evitado, y el programa continúa. Pero si se ingresa la entrada incorrecta, la bomba explota y termina el programa. Para desactivar toda la bomba, uno necesita desactivar con éxito cada uno de sus niveles.

A cada participante se le asigna una bomba única para desactivar. Su misión es aplicar sus mejores habilidades de análisis en ensamblador para descifrar la entrada requerida que les permitirá superar cada nivel y desactivar la bomba por completo..

La bomba (**bomb**) es un ejecutable, es decir, código objeto ya compilado. A partir de este código, se deberá trabajar hacia atrás en un proceso conocido como **ingeniería inversa**, para intentar reconstruir una imagen del código fuente original en C. Una vez que se comprenda cómo funciona la bomba asignada, podrá proporcionar la entrada correcta en cada nivel para desactivarla paso a paso.

Los niveles se vuelven progresivamente más complejos, pero la experiencia adquirida al avanzar en cada uno debería compensar esta dificultad. Un aspecto importante a tener en cuenta es que la bomba tiene un disparador extremadamente sensible, que puede explotar ante la menor provocación. Cada vez que la bomba

explota, se notifica al personal, lo que resulta en una reducción de puntos en el trabajo práctico. Por lo tanto, **detonar la bomba tiene consecuencias negativas**.

La **ingeniería inversa** exige una combinación de enfoques y técnicas diversas, y ofrece la oportunidad de practicar con una variedad de herramientas, que se listan y explican brevemente en la sección 4. La herramienta más poderosa en este proceso será el **debugger**, y uno de los objetivos es mejorar la destreza con **gdb**. Desarrollar un sólido dominio de **gdb** puede proporcionar grandes beneficios a lo largo de toda la carrera profesional.

2 - ¿Cómo empezar?

1. Lean **TODO** el tp.

2. Cuidado con simplemente ejecutar la bomba ya que probablemente explote. Háganlo primero usando gdb y escriban los breakpoints correspondientes para evitar que explote:

```
$gdb bomb
```

3. **bomb** puede recibir parámetros de la siguiente forma:

```
$gdb --args bomb <arg1> <arg2> ...
```

4. Impriman el assembly de la bomba así pueden hacer un mapa mental:

```
$objdump -M intel -d bomb
```

5. Les recomendamos ***fuertemente*** redirigir la salida hacia un archivo (por ejemplo, assembly_my_bomb.txt), abrirlo con un editor de texto e intenten hacer una especie de mapa del código. Es decir, intentar entender qué hace cada una de las secciones del código y alguna especie de esquema de las fases de la bomba, les va a ayudar bastante:

```
$objdump -M intel -d bomb > assembly_my_bomb.txt
```

6. ***¡Repasen las clases de gdb, pasajes de parámetros y pila! No obstante, les proveemos algunos machetes con las instrucciones de gbd, y algunos otros softwares que les pueden resultar útiles (sección 4). Igualmente, insistimos vean y estudien bien las clases.***

3- Entrega:

1. Deben subir **TODOS** los archivos proporcionados a una carpeta específica dentro de su carpeta de entrega en un repositorio de Github. La corrección es automática, por lo que cualquier archivo que se les haya dado y que NO sea incluido en la entrega resultará en la desaprobación automática del trabajo práctico. Les recomendamos ejecutar el siguiente comando para asegurarse de que todos los archivos están siendo correctamente incluidos en su repositorio:

   $git add .

   Este comando garantizará que todos los archivos del repositorio se entreguen correctamente. **Importante**: Agreguen como colaboradores a todos los docentes, completen la planilla de entregas del TP2 con link al repo y hash del commit.

Archivos a agregar dentro del repositorio de entrega:

2. **input.txt:** Archivo de inputs contiene los strings que se usan para resolver cada fase de la bomba. Tendríamos que poder correr:

   $./bomb < input.txt

   y desactivar al menos 3 fases de la bomba.

3. **respuestas_descripcion.txt:** Acá deben mencionar: **Su nombre + su email.** Luego para cada etapa que desactivaron explicar qué hacía el código, y cómo lo resolvieron. No tiene que escribir una novela, simplemente algunas oraciones describiendo el nivel y su solución.

4 - Herramientas útiles:

Aquí hay algunos posibles puntos de ataque para tu gran aventura de ingeniería inversa:

nm: volcará la tabla de símbolos de un ejecutable. Los símbolos incluyen los nombres de funciones y variables globales y sus direcciones. La tabla de símbolos por sí sola no es mucho por donde empezar, pero simplemente leer los nombres podría darte una ligera idea del terreno.

strings: mostrará todas las cadenas imprimibles en un ejecutable, incluidas todas las constantes de cadena. ¿Qué cadenas encuentras en tu bomba? ¿Alguna de ellas parece relevante para la tarea en cuestión?

objdump: puede volcar el código objeto en su equivalente desensamblado. Leer y rastrear el código desensamblado es de donde vendrá la mayor parte de tu información. Escudriñar el código objeto sin vida sin ejecutarlo es una técnica conocida como listado muerto. Una vez que averigües qué hace el código objeto, puedes, en efecto, traducirlo de vuelta a C y luego ver qué entrada se espera. Esto funciona bastante bien en pasajes de código simples, pero puede volverse complicado cuando el código es más complejo.

gcc: Si no estás seguro de cómo se traduce una construcción de C particular a ensamblador o cómo acceder a un cierto tipo de datos, otra técnica es intentar comenzar desde el otro lado. Escribe un pequeño programa en C con el código, compila y luego rastrea su desensamblado, ya sea listado en un archivo con objdump o en gdb. Por ejemplo, si no estás seguro de cómo funciona una declaración break o cómo se invoca un puntero a función por qsort, esta sería una buena manera de averiguarlo. Dado que tú mismo escribiste el programa de prueba, tampoco tienes que temer su naturaleza explosiva.

El sitio web interactivo GCC Explorer. Menos pesado que iniciar gcc por ti mismo es el práctico gcc-en-un-sitio-web que adelantamos en el laboratorio 6. Escribe un fragmento de código y obtén su traducción inmediata al ensamblador, ¡fácil! La herramienta está haciendo la misma traducción que podrías hacer a través de gcc, pero de una manera conveniente que fomenta la exploración interactiva.

5 - Algunos tips que pueden llegar a ser útiles:

Disassembly de la sección .text: (la sección de text es la de código)

objdump -d -M intel bomb

Si se desea encontrar una cadena de texto:

strings bomb

Ver la tabla de símbolos del ejecutable. (**No confundir con el comando string)**

nm bomb

Recuerden que siempre lo mejor es usar el manual, por ejemplo:

man nm

6 - Debugger

GDB Quick Reference

Comandos útiles:

r	|	**run** Ejecuta el programa hasta el primer break
b	|	**break FILE:LINE** Breakpoint en la línea
b	|	**break FUNCTION** Breakpoint en la función
info breakpoints		Muestra información sobre los breakpoints
c	|	**continue** Continua con la ejecución
s	|	**step** Siguiente linea (Into)
n	|	**next** Siguiente linea (Over)
si	|	**stepi** Siguiente instrucción asm (Into)
ni	|	**nexti** Siguiente instrucción asm (Over)

x/Nuf ADDR Muestra los datos en memoria

N = Cantidad (bytes)

u = Unidad b | h | w | g

b:byte, h:word, w:dword, g:qword

f = Formato x | d | u | o | f | a

x:hex, d:decimal, u:decimal sin signo, o:octal, f:float,

a:direcciones, s: strings, i:inst.

Ejemplos:

- x/3bx **addr** : Tres bytes en hexadecimal.
- x/5wd **addr** : Cinco enteros de 32 bits con signo.
- x/i : $rip : imprimir próxima instrucción
- x/s **addr** : Una string terminada en cero.

Configuración de GDB:

~ /.gdbinit

Para usar sintaxis intel y guardar historial de comandos (el archivo de **.gdbinit** vino con el práctico):

set disassembly-flavor intel
set history save

Correr GDB con argumentos:

gdb --args <ejecutable> <arg1> <arg2> ...

MISC:

- ¿Cómo reconozco un binario?

file bomb

En general, arroja lo siguiente:

ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, with debug_info, not stripped

ELF, siglas de ***"Executable and Linkable Format"***, es un formato de archivo utilizado para ejecutables, bibliotecas compartidas y objetos en sistemas operativos tipo Unix y Unix-like, como Linux y BSD. ELF proporciona una estructura estándar para organizar y ejecutar programas, incluyendo información sobre el tipo de archivo, segmentación de memoria, símbolos, y otras características necesarias para su ejecución.

En particular, este archivo es un programa ejecutable diseñado para sistemas Unix/Linux de 64 bits. Está creado para la arquitectura x86-64 y sigue el estándar SYSV. Está enlazado estáticamente, lo que significa que todas las bibliotecas necesarias están incluidas dentro del archivo. Además, contiene información de depuración para ayudar en el diagnóstico de errores durante el desarrollo. Es ***Not stripped***, lo que significa que conserva todos los detalles adicionales como símbolos de depuración. En resumen, es un ejecutable completo que puede ejecutarse en sistemas unix compatibles, con capacidad de depuración integrada y sin haber sido optimizado para reducir su tamaño.