TRABAJO DE FUNDAMENTOS DE COMPUTADORES Y REDES

Curso 2024-2025

OBJETIVOS DEL TRABAJO

El trabajo persigue varios objetivos:

- Aplicar los conocimientos de la asignatura en un entorno realista, aunque simplificado.
- Entrar en contacto con una arquitectura real, en concreto, la arquitectura Intel x86-32. Además, al conocer una segunda arquitectura en la asignatura, el alumno puede apreciar cómo conceptos generales se pueden aplicar de distintas maneras en implementaciones concretas.
- Practicar el lenguaje C/C++, necesario para otras partes de la asignatura y para asignaturas futuras, además de ser un lenguaje muy usado en la práctica. Además, se verá cómo mezclarlo con ensamblador.
- Practicar el uso de máscaras y desplazamientos.
- Desarrollar habilidades de trabajo en grupo y documentación básicas para un Ingeniero en Informática.

DESARROLLO DEL TRABAJO

El trabajo se desarrollará en dos fases.

FASE 1 (5 PUNTOS)

Fase 1 – primera parte (2,5 puntos)

Durante esta fase, cada grupo debe realizar un programa en C/C++ para practicar conceptos como manejo de máscaras, desplazamientos y cadenas. Para realizar este programa, se debe partir de la solución **Teamwork** que se puede descargar del Campus Virtual de la asignatura. Esta solución está configurada con ciertas opciones de compilación y enlazado que facilitan la interpretación del código generado, y que no deben ser modificadas por el alumno en ningún caso.

El programa que se va a desarrollar simula un control de licencias muy sencillo. Va a estar parametrizado para cada grupo. Se utilizará un número al que llamaremos ID y se corresponderá con el número del UO más pequeño de entre todos los componentes del grupo. Nos referiremos a cada cifra del número con ID[posicion], es decir, si ID=432981, entonces ID[0]=4, ID[1]=3, etc. Por lo tanto, si en el enunciado dice algo como “si el bit ID[0] es 1” quiere decir “si el bit 4 es 1”.

Se debe incluir en un comentario al principio del código el ID utilizado.

Se deben implementar las funciones que se describen a continuación.

ControlWithReversedStrings()

La función usa cadenas, que deben ser declaradas como vectores de caracteres de 20 posiciones. No es válido usar la clase String. Al leer una cadena, se puede suponer que tiene menos de 20 caracteres. No se llevará a cabo comprobación de errores. Ten en cuenta que esto es una mala práctica, pero como estamos empezando con C, obviaremos estas importantes comprobaciones.

La función debe seguir estos pasos:

- Leer una cadena de la terminal. Comprobar con strlen() que la longitud de la cadena es al menos 10 y que el carácter con índice ID[5] es igual al carácter con índice ID[2]. Si no, debe imprimir el mensaje “Acceso incorrecto” y llamar a exit().
- Leer otra cadena de la terminal. Usando la función strcmp(), comprobar que la inversa de la cadena es *9n7g4fd1*. Si no, debe imprimir el mensaje “El acceso no fue correcto” y llamar a exit().

MaskControl()

En primer lugar, debe pedir dos números enteros sin signo de 32 bits. A continuación, tiene que comprobar:

- Que el bit ID[2] del primer número es igual a 0. Si no, debe imprimir el mensaje "Acceso erróneo" y llamar a exit().
- Que el bit ID[5] del primer número es igual al bit ID[0] del segundo número. Si no, debe imprimir el mensaje "Intruso detectado" y llamar a exit().
- Que el número formado con los ID[3] bits de peso más alto del primer número con el resto de los bits de peso más bajo del segundo número es mayor que el valor ID[1]*100. Si no, debe imprimir el mensaje "Hubo algún fallo" y llamar a exit().

ControlInAsm()

Debe leer tres números enteros de 32 bits de la terminal y pasárselos a la función IsValidAssembly(). A continuación debe comprobar el resultado de la llamada y, si es 0, debe imprimir el mensaje "Algo salió mal" y llamar a exit().

La función IsValidAssembly() se debe implementar en ensamblador en el fichero Assembly-code.asm y debe retornar 1 si estas dos condiciones sobre los números leídos de la terminal son ciertas y 0 en caso contrario:

- El valor de los ID[1] bits más bajos del primer número interpretados en binario natural es mayor que 108.
- El bit ID[2] del segundo número es igual al bit ID[0] del tercer número.

CheckArray()

Esta función debe en primer lugar declarar un vector de 3 elementos de 8 bits cada uno. A continuación, debe leer de la terminal el valor de los tres elementos. Finalmente, debe comprobar que la operación AND a nivel de bits de los elementos es igual a 200. Si no es así, debe imprimir el mensaje Fallo y llamar a exit().

main()

Debe llamar a las funciones **ControlWithReversedStrings()**, **MaskControl()**, **ControlInAsm()** y **CheckArray()**. Si se llegan a ejecutar todas sin llamar a **exit()**, debe mostrar el mensaje "Acceso permitido".

Fase 1 – segunda parte (2,5 puntos)

A partir de la solución de Visual Studio presentada por cada grupo se debe responder a una serie de cuestiones. Para cada una de ellas **se deben explicar los pasos que se han seguido para hallar la solución, incluyendo las capturas de pantalla en las que se pueda comprobar de dónde se ha obtenido la respuesta.**

Se pide:

- (1 punto) Dirección de memoria a partir de la cual se sitúa el código de paso de parámetros a la función **IsValidAssembly()** y el propio código en sí en forma de código máquina y mnemónicos.
- (0,5 puntos) Dirección de la memoria en la que se encuentra la cadena que se lee en la primera función indicada en las instrucciones.
- (0.5 puntos) Marco de pila de la última función indicada en las instrucciones, en su estado después de leer los elementos del vector. Deberás hacer una captura de pantalla e indicar sobre ella qué son los distintos elementos que hay en memoria, como se muestra en el ejemplo para esta función **test()**:

```
void test() {
    int a = 15;
}

0x0019FEB8  0f 00 00 00 ←Variable a
0x0019FEBC  f0 fe 19 00 ←Copia de EBP
0x0019FEC0  35 10 40 00 ←Dirección de retorno
```

- (0.5 puntos) Explicación del acceso de lectura a un elemento del vector en la última función indicada en las instrucciones. Debes indicar cuál es el mnemónico que se usa para leer el valor de un elemento del vector y debes indicar qué significan las distintas partes que aparecen.

Vuestro grupo de trabajo tendrá un foro en el Campus Virtual y en él debéis realizar las entregas. Para la **Fase 1** la fecha límite de entrega es el **lunes 24 de marzo.** Se debe entregar un archivo comprimido en formato zip con el archivo de código fuente **.cpp,** el código fuente **.asm** y una memoria en PDF con los nombres y UO de los alumnos, un ejemplo de entrada válida y otro de una entrada inválida para cada función y la respuesta a las preguntas, con explicaciones y capturas de pantalla. Además, se debe indicar al final de la memoria cómo se ha repartido el trabajo entre los miembros del grupo y cuántas horas ha dedicado cada uno.

FASE 2 (5 PUNTOS)

El mundo te necesita. Una especie alienígena amenaza a la humanidad. Han situado una serie de bombas en infraestructuras críticas que sólo pueden ser desactivadas mediante unos programas llamados "bombas binarias". Estos programas tienen 4 etapas. En cada una piden una entrada. Si cumple ciertos criterios, la etapa se desactiva y se pasa a la siguiente; si no, la bomba estalla. Si se consiguen desactivar todas las etapas, la bomba será desactivada.

Los profesores de la asignatura no tenemos tiempo a desactivarlas todas, así que hemos decidido pedir la colaboración de los alumnos.

Por fortuna, una heroica incursión en los servidores alienígenas nos ha permitido obtener el código fuente del programa principal de cada bomba y su información de depuración. Tu profesor proporcionará a vuestro grupo la bomba binaria que tenéis que lograr desactivar. Para ello debéis deducir, a partir de la depuración del código, valores de entrada válidos.

Por cada etapa que logréis desactivar, obtendréis un punto. El punto final se obtendría si lográis modificar el código de la bomba para que indique que está desactivada sin necesidad de introducir ninguna entrada.

En el foro para vuestro equipo del **Campus Virtual** se debe entregar, el **lunes 28 de abril** como fecha límite, un archivo .zip con el fichero .exe modificado para que se desactive sin proporcionar ninguna entrada y una memoria en formato PDF indicando las entradas que desactivan cada etapa y el proceso que habéis seguido para obtenerlas. Se deben utilizar capturas de pantalla en las explicaciones. También se debe explicar cómo se ha logrado modificar el código para que se desactive sin necesidad de introducir ninguna entrada. Por último, se debe indicar cómo se ha repartido el trabajo entre los miembros del grupo y cuántas horas ha dedicado cada uno.

EVALUACIÓN

La evaluación se realizará en base a las puntuaciones señaladas en la sección de desarrollo del trabajo. El resultado total dependerá de la calidad del código y de la redacción y presentación de la memoria. Esta nota podrá ser ponderada por otro factor en función de la aportación de cada alumno al trabajo. Cada día de retraso en la entrega de cualquiera de las fases restará 2 puntos.

Si se detecta que se ha realizado el trabajo por medios ilícitos o ha sido realizado por otras personas, la nota en la **evaluación continua** para **todos los miembros** del grupo será de **cero**.

INDICACIONES PARA EL DESARROLLO DEL TRABAJO

Para el desarrollo de la fase 1 es imprescindible haber realizado la sesión 1 del bloque 0 de prácticas de la asignatura, donde se introduce el lenguaje de programación C/C++. También se dispondrá de documentos y vídeos de formación para poder realizar el trabajo.

ANEXO: MÁSCARAS Y DESPLAZAMIENTOS

Una máscara de bits es un conjunto de bits que se utiliza en operaciones bit a bit. Las máscaras tienen varios usos en función de con qué operación lógica se combinen:

- AND: sirven para detectar si un bit está a 0 o a 1. Por ejemplo, si se tiene una variable **v** de 8 bits y se hace la máscara con el valor 0x4 (0000 0100b), el resultado será cero si el bit 2 de **v** está a cero y 1 en caso contrario, independientemente del valor de otros bits de **v**. Esta operación en C/C++ se indica con el símbolo **&**. Ejemplo: **v & 0x4**. También sirve para poner a 0 un bit si se utiliza una máscara que tenga a 1 todos los bits menos el que interesa poner a cero. Por ejemplo, si se quiere poner a 0 el bit 2 de **v**, se puede utilizar la máscara **0xFB** (1111 1011b), así: **v = v & 0xFB**.
- OR: sirven para poner un bit a 1, ya que el OR de 1 y cualquier bit siempre da 1. En C/C++ se indica con el símbolo **|.** Por ejemplo, **v = v | 0x4** pondría el bit 2 de **v** a 1 sin cambiar el resto de bits.
- XOR: sirven para cambiar el valor de un bit de 0 a 1 o de 1 a 0. En C/C++ se indica con el símbolo **^**. Por ejemplo, **v = v ^ 0x4** pondría el bit 2 de **v** a 1 si era 0 y a 0 si era uno.

Otra operación necesaria en muchas ocasiones es desplazar los bits de una variable hacia la derecha o hacia la izquierda. Por ejemplo, para indicar el color de un pixel es común utilizar una variable de 32 bits donde se indica en los 8 bits más altos la cantidad de transparencia (alfa), en los 8 siguientes la cantidad de rojo, en los 8 siguientes la cantidad de verde y en los 8 menos significativos, la cantidad de azul, como se muestra en el ejemplo de la figura:

Transparencia	Red	Green	Blue
1111 1111	1000 0101	0011 1101	1001 1001

Si se quiere obtener la cantidad de rojo en una variable **color** se puede hacer aplicando una máscara que ponga a cero todos los bits menos los que interesan (bits 8 a 15) y luego desplazando 16 bits hacia la derecha la cantidad. Así, si **color** vale 0xFF853D99, la cantidad de rojo (0x85) se puede obtener con esta operación: **rojo = (color & 0x00FF0000) >> 16**. La operación AND con **0x00FF0000** pone a cero todos los bits excepto los que nos interesan y la operación **>> 16** desplaza el resultado 16 bits a la derecha. Con el símbolo **<<** se pueden hacer desplazamientos hacia la izquierda.

Otro uso interesante de la operación OR a nivel de bits es combinar bits de dos variables. Por ejemplo, si en los 8 bits más bajos de la variable **alfa** tenemos la cantidad de transparencia y en la variable **rgb** tenemos la cantidad de rojo, verde y azul, podemos obtener el color completo de esta manera: **color = (alfa << 24) | rgb**. Esta operación en primer lugar desplaza hacia la izquierda 24 bits la variable **alfa**, con lo que tenemos la transparencia en los 8 bits más altos; a continuación, se combina con una operación OR a nivel de bits con la variable **rgb** (estamos suponiendo que los 8 bits más altos de **rgb** están a cero; si no, habría que hacer antes una operación AND con 0x00FFFFFF para poner a cero esos bits sin modificar el resto).

En ensamblador de Intel x86, las operaciones con máscaras se hacen con las instrucciones **and**, **or** y **xor**. Para los desplazamientos hay dos tipos de instrucciones:

- Desplazamientos lógicos: **shl**, de *shift left*, para la izquierda, y **shr** para la derecha.
- Desplazamientos aritméticos: **sal**, de *shift arithmetic left*, para la izquierda y **sar** para la derecha.

Los desplazamientos lógicos y aritméticos hacia la izquierda son iguales: desplazan los bits a la izquierda e introducen ceros por la derecha. Los desplazamientos a la derecha se diferencian en que el desplazamiento lógico introduce por la izquierda ceros pero el desplazamiento aritmético introduce el valor que tenía el bit de más peso antes de realizar el desplazamiento, lo que permite conservar el signo en números representados en complemento a 2. Ejemplos con cantidades de 8 bits:

- Valor inicial: 10110111
  - Desplazamiento lógico a la derecha de 3 bits: 00010110
  - Desplazamiento aritmético a la derecha de 3 bits: 11110110
  - Desplazamiento lógico/aritmético a la izquierda de 3 bits: 10111000

- Valor inicial: 00110111
  - Desplazamiento lógico a la derecha de 3 bits: 00000110
  - Desplazamiento aritmético a la derecha de 3 bits: 00000110
  - Desplazamiento lógico/aritmético a la izquierda de 3 bits: 10111000