



Trabajo en grupo

Fase I

PL3-B

Andrés Fernández-Junquera Fernández UO302806

Bruno Martín Rivera UO302144

Javier Ortín Rodenas UO299855

Mateo Rama García UO300710

Fundamentos de computadores y redes

Índice

1. Primera parte	2
1.1. PasswordControl()	2
1.2. CountActiveBits()	2
1.3. AsmBasedControl()	2
1.4. ArrayMinMax()	3
2. Segunda parte	4
2.1. Dirección de memoria IsValidAssembly	4
2.2. Dirección de memoria PasswordControl	6
2.3. Marco de pila ArrayMinMax	7
2.4. Acceso de lectura ArrayMinMax	7
3. División del trabajo	9

1. Primera parte

1.1. PasswordControl()

En primer lugar, definimos una constante `maxChars` con valor 20 para la longitud de los arrays de caracteres que usemos. Estas cadenas solo podrán contener hasta 19 caracteres útiles debido al espacio necesario para `\0` que indica el final de la cadena.

Para la primera parte del método, declaramos un array de caracteres `input1` y almacenamos en él la primera entrada del usuario. Luego, usamos la función `strcmp` para comparar la entrada con la contraseña. Si el resultado es distinto de 0, es que son diferentes, y prohibimos la entrada.

Para la segunda parte, guardamos la nueva entrada del usuario en el array de caracteres `input2`. Luego, comprobamos si la longitud del array es menor que 15 o si no coinciden los caracteres en las posiciones 8 y 5. En caso de que no se cumpla alguna de estas condiciones, indicamos que se ha producido un fallo.

- Entradas válida: `input2 = "abcdeighijklmnop"`, `input2 = "aaaaabaabaaaaaa"`
- Entrada no válida: `input2 = "abcd"`, `input2 = "abcdefghijklmnop"`

1.2. CountActiveBits()

Esta función debe pedir dos números enteros sin signo. Posteriormente, debe contar el número de bits activos que hay en cada número entre la posición 5 y la 8, ambos inclusive. Finalmente, en caso de que el número de bits activos entre las posiciones 5 y 8 de los dos números no sea igual, la función imprimirá "No coinciden" y llamará a la función `exit()`.

- Entrada válida: `a = 352`, `b = 448`
- Entrada no válida: `a = 352`, `b = 0`

1.3. AsmBasedControl()

Esta función debe leer tres enteros y pasárselos a `IsValidAssembly`. Según la parametrización original del enunciado, esta segunda función debe comprobar que se cumplan las dos condiciones siguientes:

- El bit 8 del segundo número es igual al bit 5 del tercer número
- El valor de los 2 bits más bajos del primer número interpretados como binario natural es mayor que 11

Como la codificación de 11 en binario natural es `1011b`, la segunda condición no puede ocurrir nunca en estas condiciones. Por tanto, escribimos la función para que tome los 4 bits más bajos del primer entero, los interprete como natural, y lo compare con 11.

- Entrada válida: `a = -3`, `b = 403`, `c = 56`
- Entrada no válida: `a = 1`, `b = 7`, `c = 5`

1.4. ArrayMinMax()

Esta función crea un vector de 3 posiciones de elementos de 8 bits. Después, pide por consola los valores para asignar en el mismo, restringiendo los valores de entrada a números enteros entre -128 y 127, es decir, los valores enteros codificables con 8 bits con la codificación de complemento a 2. Si la entrada no es válida, se informará por consola y se pedirá de nuevo un valor.

A continuación, calcula y muestra por consola tanto el valor máximo como el mínimo de los elementos del vector introducidos. Si la diferencia entre estos valores no es inferior a $2 \cdot \text{ID}[2]$ (en nuestro caso $2 \cdot 9 = 18$), se mostrará por pantalla el mensaje "Fallo" y se llamará a `exit()`.

- Entrada válida: `arr[0] = -2, arr[1] = 7, arr[2] = 6`
- Entrada no válida: `arr[0] = 2, arr[1] = 5, arr[2] = -20`

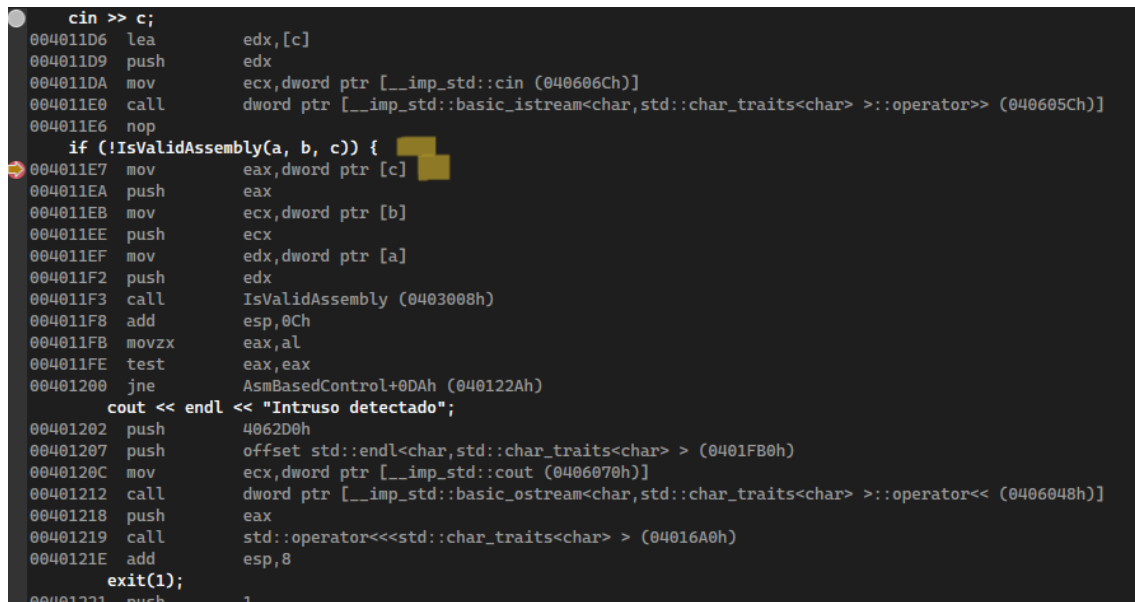
2. Segunda parte

Hemos utilizado la versión Visual Studio 2022 para la depuración.

2.1. Dirección de memoria IsValidAssembly

Para responder a la pregunta, ponemos un punto de interrupción junto antes de la llamada a la función `IsValidAssembly`. A continuación, ejecutamos el programa en modo depuración. Para poder saber las direcciones de memoria, debemos ir a **Depurar** → **Ventanas** → **Desensamblado**. En esta ventana, podemos ver las direcciones de memoria a partir de las cuales se sitúa el código de paso de parámetros a la función `IsValidAssembly`.

Buscamos la línea de código fuente en la que se realiza la llamada a dicha función. Debajo de la llamada en C++, nos encontramos con la primera dirección de memoria a partir de la cuál se sitúa el código de paso de parámetros a la función en ensamblador al apilar los registros correspondientes (se apilan los parámetros de derecha a izquierda). En nuestro caso, la dirección de memoria es **004011E7**, como se puede observar en la siguiente imagen.



```
cin >> c;
004011D6 lea     edx,[c]
004011D9 push    edx
004011DA mov     ecx,dword ptr [__imp_std::cin (040606Ch)]
004011E0 call   dword ptr [__imp_std::basic_istream<char,std::char_traits<char> >::operator>> (040605Ch)]
004011E6 nop
if (!IsValidAssembly(a, b, c)) {
004011E7 mov     eax,dword ptr [c]
004011EA push    eax
004011EB mov     ecx,dword ptr [b]
004011EE push    ecx
004011EF mov     edx,dword ptr [a]
004011F2 push    edx
004011F3 call   IsValidAssembly (0403008h)
004011F8 add     esp,0Ch
004011FB movzx   eax,al
004011FE test   eax,eax
00401200 jne     AsmBasedControl+0DAh (040122Ah)
cout << endl << "Intruso detectado";
00401202 push    4062D0h
00401207 push    offset std::endl<char,std::char_traits<char> > (0401FB0h)
0040120C mov     ecx,dword ptr [__imp_std::cout (0406070h)]
00401212 call   dword ptr [__imp_std::basic_ostream<char,std::char_traits<char> >::operator<< (0406048h)]
00401218 push    eax
00401219 call   std::operator<<<std::char_traits<char> > (04016A0h)
0040121E add     esp,8
exit(1);
00401221 push    1
```

Para poder ver el código máquina y los mnemónicos de la función `IsValidAssembly`, debemos realizar la ejecución hasta llegar al punto de interrupción que hemos puesto antes de la llamada a dicha función. Una vez llegamos a este punto, pulsamos la tecla F11 para ir al código en ensamblador. A continuación, pulsamos click derecho y escogemos la opción **Mostrar bytes de código**. Así, podemos ver tanto los mnemónicos como las instrucciones en código máquina codificado en hexadecimal de la función `IsValidAssembly`, como se muestra en las siguientes imágenes.

```

;Prologo
push ebp
00403008 55          push     ebp
mov ebp, esp
00403009 8B EC      mov     ebp,esp

;Salvaguarda de registros
push edx
0040300B 52          push     edx
push ebx
0040300C 53          push     ebx
push ecx
0040300D 51          push     ecx

;Acceso a parámetros (están apilados de derecha a izquierda)
;Como a, b y c son enteros de 32bits, son de doble palabra
mov edx, [ebp + 8]; edx = a
0040300E 8B 55 08    mov     edx,dword ptr [ebp+8]
mov ebx, [ebp + 12]; ebx = b
00403011 8B 5D 0C    mov     ebx,dword ptr [ebp+0Ch]
mov ecx, [ebp + 16]; ecx = c
00403014 8B 4D 10    mov     ecx,dword ptr [ebp+10h]

;Cuerpo del procedimiento

;Comparación del bit 8 de b y el bit 5 de c
shr ebx, 8
00403017 C1 EB 08    shr     ebx,8
and ebx, 1; obtenemos el bit 8 de b
0040301A 83 E3 01    and     ebx,1
shr ecx, 5
0040301D C1 E9 05    shr     ecx,5
and ecx, 1; obtenemos el bit 8 de c
00403020 83 E1 01    and     ecx,1
cmp ebx, ecx
00403023 3B D9      cmp     ebx,ecx
jne falso; Salta si los bits son distintos
00403025 75 0F      jne     falso (0403036h)

;Interpretación de los últimos 4 bits de a en binario natural
and edx, 15; 15d = 1111d, toma los 4 bits más bajos
00403027 83 E2 0F    and     edx,0Fh
cmp edx, 11

```

```

cmp edx, 11
0040302A 83 FA 0B    cmp     edx,0Bh
jbe falso; Salta si a <= 11 interpretando a como natural
0040302D 76 07      jbe     falso (0403036h)

;Si es cierto (no ha saltado)
mov eax, 1; Deja el valor 1 en el registro de estado
0040302F B8 01 00 00 00 mov     eax,1
jmp cierto; Evita que se sobrescriba como falso
00403034 EB 05      jmp     falso+5h (040303Bh)

falso:
mov eax, 0; Deja el valor 0 en el registro de retorno
00403036 B8 00 00 00 00 mov     eax,0

cierto:

;Restauración de registros
pop ecx
0040303B 59          pop     ecx
pop ebx
0040303C 5B          pop     ebx
pop edx
0040303D 5A          pop     edx

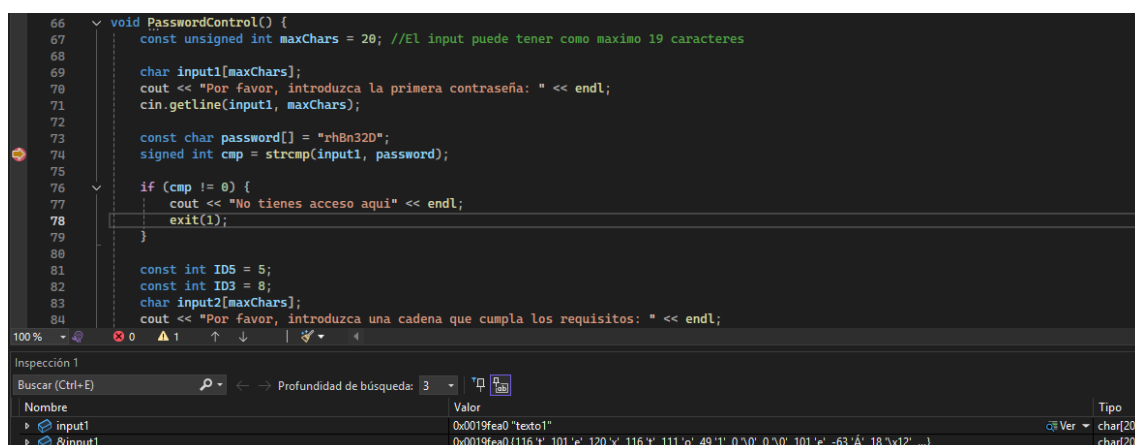
;Epílogo
pop ebp
0040303E 5D          pop     ebp
ret
0040303F C3          ret

```

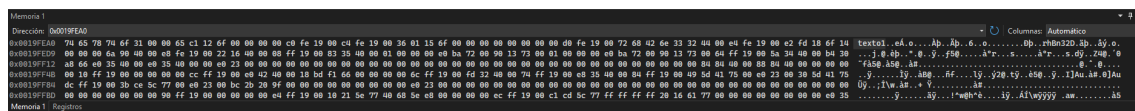
2.2. Dirección de memoria PasswordControl

La pregunta no hace referencia de manera concreta a ninguna de las dos cadenas que se leen en la función `PasswordControl`. Por tanto, hemos decidido mostrar ambas direcciones de memoria.

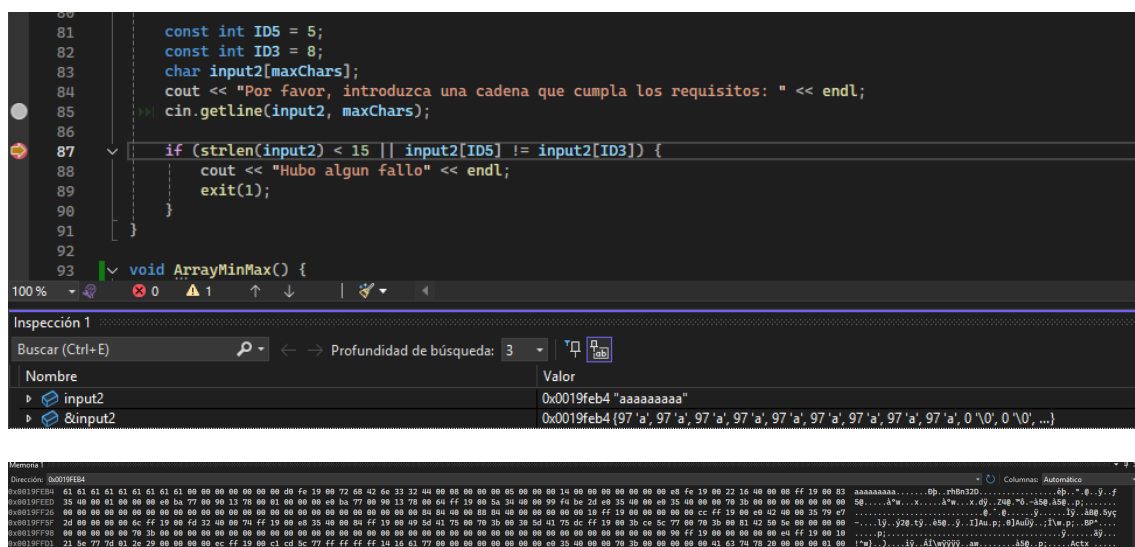
Para saber la dirección de memoria de la primera cadena, debemos poner un punto de ejecución después de leerla por terminal. A continuación, ejecutamos el programa en modo depuración y hacemos click en la opción **Ventanas** → **Depurar** → **Inspección**. A continuación, añadimos a Inspección la variable que contiene la cadena de caracteres y `&a` para saber así la dirección de memoria de la cadena. En nuestro caso, obtenemos el siguiente resultado:



Para verlo en memoria, debemos ir a **Depurar** → **Ventanas** → **Memoria**. En esta ventana, introducimos `&a`, para así poder verlo almacenado en memoria, tal y como se muestra en la siguiente imagen.

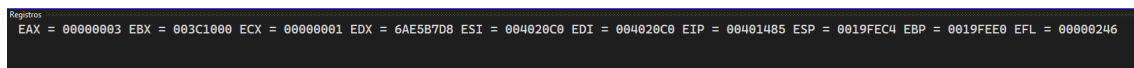


Repetimos el proceso análogamente para la segunda cadena, obteniendo lo siguiente:



2.3. Marco de pila ArrayMinMax

Para saber el marco de pila de la función `ArrayMinMax`, debemos poner un punto de interrupción después de leer los elementos del vector. A continuación, ejecutamos el programa en modo depuración y seleccionamos la opción **Ventanas** → **Depurar** → **Memoria** y **Ventanas** → **Depurar** → **Registro**. En la ventana de memoria, introducimos la dirección de memoria de la variable que contiene el vector; y en la ventana de registro, vemos en qué dirección de memoria se encuentra el puntero de pila.



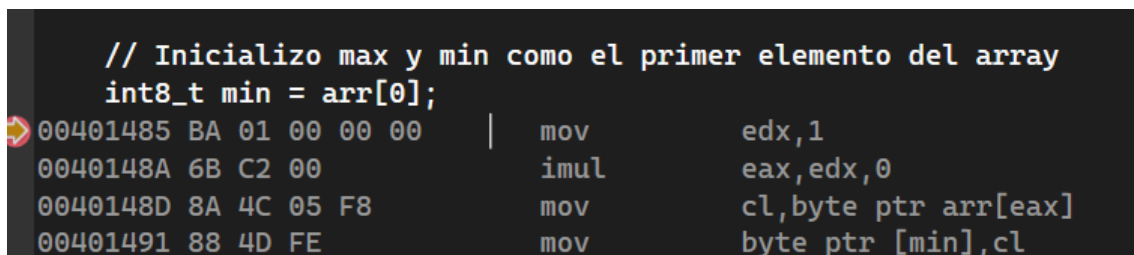
En nuestro caso, tenemos que la dirección de memoria de la variable que contiene el vector es **0x0019FED8**, la dirección de memoria del puntero de pila es **0x0019FEE0**, y la dirección de retorno es **0x0019FEE4**. Los resultados fueron interpretados a partir de la siguiente imagen:



Nótese que hay un espacio vacío para futuras variables locales que se van a usar más adelante en la ejecución del método.

2.4. Acceso de lectura ArrayMinMax

Para explicar el acceso de lectura a un elemento del vector, debemos poner un punto de interrupción en la lectura de sus elementos. En nuestro caso, cuando se lee el elemento cero del vector para inicializar al máximo. A continuación, ejecutamos el programa en modo depuración y, haciendo click derecho sobre el código, vamos al desensamblado. Como se Imagen del desensamblado obtenido:



Explicación de la imagen:

1. En la primera línea, asigna el valor 1 al registro `edx`.
2. En la segunda línea, multiplica el valor del registro `edx` (en nuestro caso el 1) por el valor 0 (ya que estamos accediendo al primer elemento del vector) y asigna el resultado al registro `eax`.

3. Mueve la información almacenada en la posición `eax` del vector (en nuestro caso es la posición 0) a los 8 bits más bajos del registro `c`. Se omite la letra `ecx` de `e` para acceder a la palabra más baja del registro, y ponemos `l` en lugar de `x` para acceder a los 8 bits más bajos, de ahí que se escriba `cl` en lugar de `ecx`.
4. Mueve la información almacenada en la parte baja del registro `ecx` a la variable local de un byte denominada `min`.

3. División del trabajo

A la hora de organizar el trabajo en equipo, nos reunimos todos para aportar ideas y pensar en común los cuatro métodos. Después de plantear varias ideas y ponernos de acuerdo en cómo hacer cada método, nos dividimos la creación de código de la siguiente manera, siempre siguiendo el guion que habíamos acordado en un principio:

- Andrés Fernández-Junquera Fernández: `ArrayMinMax()`
- Bruno Martín Rivera: `PasswordControl()`
- Javier Ortín Rodenas: `AsmBasedControl()`
- Mateo Rama García: `CountActiveBits()`

Cada alumno programó su respectiva función de manera independiente y realizó una serie de pruebas para comprobar el correcto funcionamiento del código.

A continuación, trabajamos en conjunto para responder a las cuestiones, explicar el código, y redactar la memoria. También, se han contabilizado las horas de trabajo de cada uno de los integrantes del grupo, siendo estas las siguientes:

- Andrés Fernández-Junquera Fernández: 4 horas
- Bruno Martín Rivera: 4 horas
- Javier Ortín Rodenas: 5 horas y media
- Mateo Rama García: 5 horas y media