Informe de Investigación

Lenguaje ensamblador desde 0

DE USO PÚBLICO

Un poco acerca del lenguaje ensamblador

El lenguaje ensamblador actúa como un intermediario entre el código máquina y los lenguajes de alto nivel, ofreciendo una comunicación más directa con el hardware. Cada familia de procesadores tiene su propio conjunto de instrucciones para realizar diversas operaciones, como recibir información del teclado, mostrar información en la pantalla y realizar otras tareas. Este conjunto de instrucciones se denomina "***instrucciones en lenguaje de máquina***".

Un procesador solo entiende instrucciones en lenguaje de máquina, que son cadenas de **1** y **0**. Sin embargo, el lenguaje de máquina es demasiado oscuro y complejo para su uso en el desarrollo de software. Por ello, el lenguaje ensamblador de bajo nivel está diseñado para una familia específica de procesadores que representa diversas instrucciones en código simbólico y de una forma más comprensible.

Ventajas del lenguaje ensamblador:

- Cómo se representan los datos en la memoria y otros dispositivos externos.
- Cómo el procesador accede y ejecuta las instrucciones.
- Cómo las instrucciones acceden y procesan los datos.
- Requiere menos memoria y tiempo de ejecución.

Uso del lenguaje ensamblador en la ciberseguridad:

- **Análisis de Malware:** Los analistas de malware a menudo necesitan desensamblar y analizar el código malicioso para entender su funcionamiento. Esto requiere la habilidad de leer y comprender código en ensamblador.
- **Explotación de Vulnerabilidades:** Muchos exploits, especialmente aquellos que interactúan directamente con el sistema operativo o el hardware, están escritos en ensamblador. Conocer este lenguaje permite a los profesionales de seguridad entender y desarrollar exploits.

- **Ingeniería Inversa:** La ingeniería inversa de software, que a menudo implica convertir el código binario de una aplicación de vuelta a ensamblador, es una técnica crucial para entender y analizar programas sin acceso al código fuente.

Configuración del entorno local

Una vez dicho lo anterior, tenemos que proceder a montar nuestro entorno de trabajo donde comenzaremos el inhóspito camino de aprender lenguaje ensamblador, lo cual consta de tener los siguientes requisitos:

- Una computadora.
- Sistema operativo GNU/Linux.
- Tener instalado el ensamblador NASM (Netwide Assembler) que nos ofrece:
  1. Es gratis.
  2. Contiene bastante documentación en el internet.
  3. Se puede utilizar tanto en Linux como en Windows.

También hay otras alternativas de ensamblador como:

1. Ensamblador de Microsoft (MASM).
2. Ensamblador Borland Turbo (TASM).
3. El ensamblador GNU (GAS).

NOTA: Es importante recalcar que el lenguaje ensamblador va cambiando cierta estructura, dependiendo del programa ensamblador que se elija.

Instalación del NASM

El programa ya en linux debe venir instalado y para verificar que lo este, abrimos una terminal y colocamos:

1. `whereis nasm` y presione ENTER.
2. Si ya está instalado, aparecerá una línea como nasm: `/usr/bin/nasm`.

En caso de no estar instalado:

- Abra una consola, y posteriormente se debe dirigir a la página oficial de NASM para obtener la última versión en el siguiente enlace:
  - ✔ https://www.nasm.us/pub/nasm/releasebuilds/2.16.03/
- Descargue el archivo fuente de Linux nasm-X.XX.ta.gz, donde X.XX se encuentra el número de versión de NASM en el archivo.

- Descomprima el archivo en un directorio que crea un subdirectorio **nasm-X. XX**.
- Haga clic en cd **nasm-X.XX** y escriba ./configure. Este script de shell encontrará el mejor compilador de C para usar y configurará los makefiles en consecuencia.
- Escriba make para crear los binarios nasm y ndisasm.
- Escriba **make install** para instalar nasm y ndisasm en **/usr/local/bin** y para instalar las páginas del manual.

Sintaxis básica

El lenguaje ensamblador se divide en (3) secciones tal como:

- La sección de datos.
- La sección bss, y.
- La sección de texto.

La sección de datos

La sección de datos se utiliza para declarar datos inicializados o constantes. Estos datos no cambian en tiempo de ejecución. En esta sección, puede declarar varios valores como constantes, nombres de archivos o tamaño de búfer, etc. La sintaxis para declarar la sección de datos es:

```
section.data
```

La sección bss

La sección bss se utiliza para declarar variables. La sintaxis para declarar la sección bss es:

```
section.bss
```

La sección de texto

La sección de texto se utiliza para guardar el código real. Esta sección debe comenzar con la declaración **global** _**start**, que le indica al núcleo dónde comienza la ejecución del programa. La sintaxis para declarar la sección de texto es:

```
section.text
   global _start
_start:
```

Comentarios

El comentario en lenguaje ensamblador comienza con un punto y coma (;). Puede contener cualquier carácter imprimible, incluido un espacio en blanco. Puede aparecer solo en una línea, como −

```
; Este es un comentario
```

o, en la misma línea junto con una instrucción, como −

```
add eax, ebx ; adds ebx to eax
```

Declaraciones en lenguaje ensamblador

Los programas en lenguaje ensamblador constan de tres tipos de declaraciones:

- **Instrucciones ejecutables o instrucciones:** Le indican al procesador qué hacer. Cada instrucción consta de un código de operación (opcode). Cada instrucción ejecutable genera una instrucción en lenguaje de máquina.
- **Directivas de ensamblador o pseudo-operaciones:** Le informan al ensamblador sobre los distintos aspectos del proceso de ensamblaje. No son ejecutables y no generan instrucciones en lenguaje de máquina.
- **Macros:** Son básicamente un mecanismo de sustitución de texto.

Sintaxis de las sentencias en lenguaje ensamblador

Las instrucciones en lenguaje ensamblador se introducen una instrucción por línea. Cada instrucción sigue el siguiente formato:

```
[Etiqueta]  mnemónico  [operadores]  [;comentario]
```

Los campos entre corchetes son opcionales. Una instrucción básica tiene dos partes: la primera es el nombre de la instrucción (o el mnemónico) que se va a ejecutar y la segunda son los operando o parámetros del comando. A continuación se muestran algunos ejemplos de declaraciones típicas en lenguaje ensamblador:

Segmentos de la memoria

Un modelo de memoria segmentada divide la memoria del sistema en grupos de segmentos independientes a los que se hace referencia mediante punteros ubicados en los registros de segmentos. Cada segmento se utiliza para contener un tipo específico de datos.

Un segmento se utiliza para contener códigos de instrucciones, otro segmento almacena los elementos de datos y un tercer segmento mantiene la pila del programa. A continuación los (3) segmentos del lenguaje ensamblador:

- **Segmento de datos:** Está representado por la sección .data y el .bss . La sección .data se utiliza para declarar la región de memoria, donde se almacenan los elementos de datos para el programa. Esta sección no se puede expandir después de que se declaran los elementos de datos y permanece estática durante todo el programa. La sección .bss también es una sección de memoria estática que contiene búferes para los datos que se declararán más adelante en el programa. Esta memoria intermedia se llena con ceros.
- **Segmento de código:** Está representado por la sección .text . Define un área en la memoria que almacena los códigos de instrucción. También es un área fija.
- **Pila:** En este segmento de memoria, los datos pueden inicializarse en tiempo de ejecución. Actúa como almacenamiento temporal y sigue el protocolo Last In First Out , lo que significa que los elementos solo se agregan o eliminan desde arriba.

Registros del procesador

Antes de iniciar con los registros es importante hacer un repaso sobre el equivalencia de bit y bytes.

Bytes y bit

8 Bits = 1 byte
1 byte = “1” o “0”
8 bits = 11001100
1 Byte = 11001100

Tamaño en registros:

- 8 Byte = 1 Byte > max = 255d
- 16 bits = 2 Bytes > max = 65535d
- 32 bits = 4 Bytes > max = 4294967295d

División de los registros:

- EAX = 32 bits = 1111 1111 1111 1111 1111 1111 1111 1111
- AX = 16 bits = 1111 1111 1111 1111
- AH (Hight) = 8 bits = 1111 1111
- AL (Low) = 8 bits = 1111 1111

Errores de combinaciones:

- 8 bits + 8 bits = Ejecuta
- 16 bits + 16 bits = Ejecuta
- 32 bits + 32 bits = Ejecuta

- 16 bits + 8 bits = Error
- 16 bits + 32 = Error
- 32 bits + 8 bist = Error

Los registros son ubicaciones de almacenamiento de alta velocidad, dichos registros se encuentran alojados en el procesador y cuentan con una velocidad de acceso mayor que la memoria convencional. Los procesadores x86 cuentan con una serie de registros disponibles para utilizar como almacenamiento temporal para variables, valores y demás información que utilizan durante la ejecución de instrucciones como así también punteros a secciones de memoria como la pila. Existen (04) diferentes registro, tales como:

- Registros generales.
- Segmentos de registros.
- Flags (banderas de estado).
- Instruction Pointer (IP), puntero a la próxima instrucción a ejecutar.

Registros generales	Segmentos de registros	Registros de estado	Instruction Pointer (IP)
EAX (AX, AH, AL)	CS	EFLAGS	EIP
EBX (BX, BH, BL)	SS		
ECX (CX, CH, CL)	DS		
EDX (DX, DH, DL)	ES		
EBP (BP)	FS		
ESP (SP)	GS		
ESI (SI)			

Equivalencia de los registros

Nombres de 32 bits	Nombres de 16 bits	Nombres de 16 bits	Nombres de 8 bits	
EAX		**AX**	**AH AL**	Acumulador
EBX		**BX**	**BH BL**	Índice base
ECX		**CX**	**CH CL**	Conteo
EDX		**DX**	**DH DL**	Datos
ESP		**SP**		Apuntador de la pila
EBP		**BP**		Apuntador de la base
EDI		**DI**		Índice destino
ESI		**SI**		Índice de origen

EIP		**IP**		Apuntador de instrucciones
EFLAGS		**FLAGS**		Banderas

CS		Código
DS		Datos
ES		Extra
SS		Pila
FS		
GS		

Algunos de estos registros de datos tienen un uso específico en operaciones aritméticas.

- **AX:** es el acumulador principal se utiliza en la entrada/salida y en la mayoría de las instrucciones aritméticas. Por ejemplo, en la operación de multiplicación, un operando se almacena en el registro EAX o AX o AL según el tamaño del operando.
- **BX:** se conoce como registro base, ya que podría usarse en direccionamiento indexado.
- **CX:** se conoce como registro de conteo, al igual que ECX, los registros CX almacenan el conteo del bucle en operaciones iterativas.

- **DX:** se conoce como registro de datos. También se utiliza en operaciones de entrada/salida. También se utiliza con el registro AX junto con DX para operaciones de multiplicación y división que involucran valores grandes.

Los registros de se dividen en (02) categorías en registros multipropósito y registros de propósito especial.

Registros multipropósito

- **EAX (Acumulador):** Es el registro que se utiliza para la multiplicación y la división, también es el registro que por omisión recibe un numero decimal ingresado desde teclado.
- **EBX (Índice base):** Guarda la dirección de desplazamiento de una posición en el sistema de memoria.
- **ECX (Conteo):** Este registro es utilizado es utilizado como contador por el CPU, al utilizar un bucle en nuestros programas, ECX se decrementa según la cantidad de repeticiones que se haya asignado, normalmente se decrementa un valor por repetición, aunque se puede programar para decrementarse más rápido.
- **EDX (Datos):** Almacena una parte del resultado de una multiplicación o parte del dividendo antes de una división.
- **EDI (Índice de destino) y ESI (Índice de origen):** Ambos registros se utilizan para instrucciones de transferencia de memoria de alta velocidad.
- **ESP:** Es utilizado raras veces para realizar operaciones aritméticas o para movimientos de memoria, su mayor función es direccionar la pila. Es conocido también como:
- **Apuntador de pila extendido EBP** o también llamado apuntador de estructura extendido, a diferencia de ESP, EBP no debe utilizado para operaciones aritméticas, su mayor función es hacer referencia a los parámetros de funciones y a las variables locales de la pila.

Registro multipropósito especial

- **EFLAGS (Bandera):** Es un registro de estado para indicar si cumple o no algún estado. Existen diferentes tipos de banderas, entre ella encontramos:
  - **ZF (Zero Flag)**: Este bit se activa cuando el resultado de una operación es igual a cero.
  - **CF (Carry Flag):** Este bit se activa cuando el resultado de una operación es muy grande o muy pequeño para el operador de destino.
  - **SF (Sign Flag):** Según si el resultado de una operación es un valor positivo o negativo. Si el valor es positivos se queda en cero y es uno en caso contrario.
  - **TF (Trap Flag):** Este flag se utiliza para depurar (debugging) un programa. En caso de que esté activo el procesador ejecutará una instrucción a la vez.

Instrucciones y Operadores

En esta sección es importante definir **¿Qué son los mnemónicos?.** Y son la abreviación de una palabra que aún así estando abreviada se le entiende su significado.

Instrucciones

Son la acciones básicas que puede realizar un procesador. Cada instrucción tiene un código de operación (**opcode**) que indica al procesador que operación realizar, en conjunto de uno o más operandos que especifican los datos sobre lo que se realizará la operación. Entre ellos encontramos:

a) **MOV:** Tiene como propósito la transferencia de datos entre registros de procesador o/u registro y memoria. Adicionalmente **mov** también permite el uso de datos absolutos, como por ejemplo mover el número 10 a un registro del procesador.

b) **ADD:** Es una instrucción aritmética que suma dos registros y almacena el resultado en el registro de destino. ADD no utiliza el flag C.

c) **SUB:** Es una instrucción que resta enteros. El resultado de restar el valor del primer operando del valor del segundo operando se almacena en el valor del segundo operando.

d) **JMP:** Se utiliza para desviar el flujo de un programa sin tomar en cuenta las condiciones actuales de las banderas ni de los datos.

e) **CALL:** Es una instrucción que resta enteros. El resultado de restar el valor del primer operando del valor del segundo operando se almacena en el valor del segundo operando.

Operadores

Se utilizan dentro de las instrucciones para especificar cómo se manipularán los datos.

- **Aritméticos: ADD, SUB, MUL, DIV.** Como los usaríamos seria así:
  - **ADD**

```
mov ax, 500d
mov bx, 500d
add  ax, bx (ax = ax + bx)
```

- **SUB**

  mov ax, 500d

  mov bx, 500d

  sub bx, ax (bx = bx - ax)

- **MUL**

  mov ax, 500d

  mov bx, 500d

  mul ax (ax = ax * bx)

- **DIV**

  mov ax, 2d

  mov bx, 500d

  div ax (ax = ax / bx)

- **Lógicos:** "**Y**" AND, **"OR"** "O", "**NOT**" "Negación", **"XOR"** "O exclusivo"
  - **AND**

    mov ax, 11001100b

    and ax, 00110011b
  - **OR**

    mov ax, 11010110b

    or ax, 01011011b
  - **NOT**

    mov ax, 11010110b

    not ax
  - **XOR**

    mov ax, 11010110b

    xor ax, 01011011b
- **Relacionales**

Es primordial destacar la diferencia de estos, ya que las instrucciones son más abstractas, ya que representan acciones a nivel de máquina. Los operadores, por otro lado, son más específicos y se utilizan dentro de las instrucciones para realizar operaciones concretas.

En cuanto a su funcionalidad las instrucciones definen la operación en general, mientras que los operadores especifican cómo se llevarán a cabo los cálculos o manipulaciones de datos dentro de esa instrucción. Si bien están estrechamente relacionados, operadores e instrucciones no son lo mismo.

`MOV EAX, [EBX]` **`;`** MOV mueve el contenido de la dirección de memoria apuntada por EBX al registro EAX

`ADD EAX, 5` **`;`** suma 5 al contenido del registro EAX

Explicación:

En la primera línea, ***MOV*** es la instrucción y ***[EBX]*** es el operando (una dirección de memoria). En la segunda línea, ***ADD*** es la instrucción, ***EAX*** es el operando de destino y 5 es el operando fuente. El símbolo + es el operador que indica la suma.

Llamadas al sistema Linux

Las llamadas del sistema son API para la interfaz entre el espacio del usuario y el espacio del núcleo. Ya hemos utilizado las llamadas del sistema sys_write y sys_exit para escribir en la pantalla y salir del programa, respectivamente.

Puede utilizar las llamadas del sistema Linux en sus programas de ensamblaje. Para utilizar las llamadas del sistema Linux en su programa, debe seguir los siguientes pasos:

- Coloque el número de llamada del sistema en el registro EAX.
- Almacene los argumentos de la llamada al sistema en los registros EBX, ECX, etc.
- Llame a la interrupción correspondiente (80h).
- El resultado normalmente se devuelve en el registro EAX.

Hay seis registros que almacenan los argumentos de la llamada al sistema utilizada. Estos son EBX, ECX, EDX, ESI, EDI y EBP. Estos registros toman los argumentos consecutivos, comenzando con el registro EBX. Si hay más de seis argumentos, la ubicación de memoria del primer argumento se almacena en el registro EBX. El siguiente fragmento de código muestra el uso de la llamada del sistema sys_exit −

```
mov  eax,1          ; system call number (sys_exit)
int  0x80       ; call kernel
```

El siguiente fragmento de código muestra el uso de la llamada del sistema sys_write −:

```
mov  edx,4
mov  ecx,msg
mov  ebx,1
mov  eax,4
int  0x80
```

Todas las llamadas al sistema se enumeran en /usr/include/asm/unistd.h , junto con sus números (el valor que se debe poner en EAX antes de llamar a int 80h). La siguiente tabla muestra algunas de las llamadas del sistema utilizadas en este tutorial:

%eax	Nombre	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	int				
2	sys_fork	struct pt_regs				
3	sys_read	unsigned int	char *	size_t		
4	sys_write	unsigned int	const char *	size_t		
5	sys_open	const char *	int	int		
6	sys_close	unsigned int				

Ejemplo de como utilizar las llamadas del sistema:

```
section .data
     userMsg db 'Please enter a number: '
     lenUserMsg equ $-userMsg
     dispMsg db 'You have entered: '
     lenDispMsg equ $-dispMsg

section .bss
     num resb 5

section .text
     global _start

_start:
     mov eax, 4
     mov ebx, 1
     mov ecx, userMsg
     mov edx, lenUserMsg
     int 80h

;Read and store the user input
     mov eax, 3
     mov ebx, 2
     mov ecx, num
     mov edx, 5 ;5 bytes (numeric, 1 for sign) of that
```

```
information
        int 80h

  ;Output the message 'The entered number is: '
        mov eax, 4
        mov ebx, 1
        mov ecx, dispMsg
        mov edx, lenDispMsg
        int 80h

  ;Output the number entered
        mov eax, 4
        mov ebx, 1
        mov ecx, num
        mov edx, 5
        int 80h

  ; Exit code
        mov eax, 1
        mov ebx, 0
        int 80h
```

Variables

En lenguaje ensamblador, el concepto de variable no es exactamente el mismo que en lenguajes de alto nivel como Python, Java o C. En ensamblador, estamos trabajando a un nivel mucho más cercano al hardware, y las variables se representan de una manera más directa. En lugar de un nombre descriptivo como en otros lenguajes, una variable en ensamblador suele ser una etiqueta que se asigna a una dirección de memoria específica y la variable en sí misma es esa dirección de memoria para reservar e inicializar uno o más Bytes. Cuando se utiliza el nombre de la variable en una instrucción, se está haciendo referencia al valor almacenado en esa dirección.

NASM ofrece varias directivas de definición para reservar espacio de almacenamiento para variables. La sintaxis para la declaración de asignación de almacenamiento para datos inicializados es:

[nombre de variable] Definición de directiva valor inicial [,valor inicial]

Donde, nombre-de-variable es el identificador de cada espacio de almacenamiento. El ensamblador asocia un valor de desplazamiento para cada nombre de variable definido en el segmento de datos. Hay cinco formas básicas de la directiva define:

Directiva	Objetivo	Espacio de almacenamiento
.section	Define el inicio de una sección	
.data y .bss	Declarar variables adentro de una sección	
.text	Indica el inicio de la sección de código	
DB	Define los byte	
DW	Define la palabra	
DD		
DQ		
DT		

Constantes

Una constante es un valor fijo que no cambia durante la ejecución del programa. Las constantes se utilizan para hacer que el código sea más legible y fácil de mantener. Existen varios tipos de constantes entre ellas encontramos:

- **Constantes Numéricas:** Valores numéricos enteros que no cambian.
  - **Ejemplo: TEN equ 10.**
  - **Flotantes:** Valores numéricos con decimales.
    - **Ejemplo: PI equ 3.14159.**
- **Cadenas de Texto:** Secuencias de caracteres que permanecen inmutables.
  - **Ejemplo: HELLO_MSG equ "Hello, World!".**
- **Constantes de Dirección:** Punteros a ubicaciones específicas en la memoria.
  - **Ejemplo: BUFFER_START equ 0x1000**
- **Constantes Simbólicas**: Nombres simbólicos que representan valores o direcciones.
  - **Ejemplo: MAX_SIZE equ 256.**
- **Constantes de Máscara:** Valores utilizados para operaciones de bits.
  - **Ejemplo: MASK equ 0xFF.**
- **Constantes de Configuración:** Valores que configuran el comportamiento del programa.
  - **Ejemplo: BAUD_RATE equ 9600.**

Condicionales

La ejecución condicional en lenguaje ensamblador se logra mediante varias instrucciones de bucle y ramificación. Estas instrucciones pueden cambiar el flujo de control en un programa. La ejecución condicional se observa en dos escenarios:

- **Unconditional jump (Salto incondicional):** Esto se realiza mediante la instrucción JMP. La ejecución condicional a menudo implica una transferencia de control a la dirección de una instrucción que no sigue a la instrucción que se está ejecutando actualmente. La transferencia de control puede ser hacia adelante, para ejecutar un nuevo conjunto de instrucciones, o hacia atrás, para volver a ejecutar los mismos pasos.
- **Conditional jump (Salto condicional):** Esto se realiza mediante un conjunto de instrucciones de salto j<condición> según la condición. Las instrucciones condicionales transfieren el control interrumpiendo el flujo secuencial y lo hacen modificando el valor de desplazamiento en IP.

Instrucción CMP

La instrucción CMP compara dos campos de datos numéricos. El operando de destino puede estar en un registro o en la memoria. El operando de origen puede ser un dato constante (inmediato), un registro o la memoria. Generalmente se utiliza en la ejecución condicional. Esta instrucción básicamente resta un operando del otro para comparar si los operandos son iguales o no. No altera los operandos de destino o de origen. Se utiliza junto con la instrucción de salto condicional para la toma de decisiones.

Sintaxis:

```
CMP destino, fuente
```

Ejemplo:

```
CMP DX,   00
JE  L7
```

El CMP se utiliza a menudo para comparar si un valor de contador ha alcanzado la cantidad de veces que se debe ejecutar un bucle. Considere la siguiente condición típica:

```
INC  EDX
CMP  EDX, 10
JLE  LP1
```

Programando “Hola Mundo”

En el siguiente código haremos un “hola mundo” en lenguaje ensamblador:

```
section   .text         ;comentario del código
     global _start

_start:
     mov  edx,len
     mov  ecx,msg
     mov  ebx,1
     mov  eax,4
     int  0x80

     mov  eax,1
     int  0x80

section   .data
msg db 'Hello, world!', 0xa
len equ $ - msg
```

Explicación del código:

Sección .text

Esta sección define el segmento de código, donde se encuentran las instrucciones que el procesador ejecutará.

_start:

Esta etiqueta marca el punto de entrada del programa, es decir, donde comienza la ejecución.

mov edx, len

- **mov:** Mueve un valor a un registro.
- **edx:** Registro de propósito general.
- **len:** Variable que contiene la longitud del mensaje.

Esta instrucción copia la longitud del mensaje a el registro edx.

`mov ecx, msg`

- **ecx:** Otro registro de propósito general.
- **msg:** Variable que contiene el mensaje a imprimir.

Aquí se copia la dirección del mensaje al registro ecx.

`mov ebx, 1`

- **ebx:** Registro de propósito general.
- **1:** Valor constante.

Se asigna el valor 1 al registro ebx, que representa el descriptor de archivo estándar de salida (stdout), generalmente la consola.

`mov eax, 4`

- **eax:** Registro de propósito general.
- **4:** Valor constante.

Se asigna el valor 4 al registro eax, que indica el número de sistema para la llamada al sistema write.

`int 0x80`

Esta instrucción interrumpe al sistema operativo y le solicita realizar la llamada al sistema especificada en los registros eax, ebx, ecx y edx. En este caso, se llama a la función write para imprimir el mensaje en la consola.

`mov eax, 1`

Se asigna el valor 1 al registro eax, que indica el número de sistema para la llamada al sistema exit.

`int 0x80`

Se interrumpe al sistema operativo nuevamente, esta vez para finalizar el programa.

`Sección .data`

Esta sección define el segmento de datos, donde se almacenan los datos utilizados por el programa.

```
msg db 'Hello, world!', 0xa
```

- **msg:** Etiqueta que identifica la variable.
- **db:** Directiva que define un byte de datos.
- **'Hello, world!', 0xa':** Cadena de caracteres a imprimir, seguida del carácter de nueva línea.

```
len equ $ - msg
```

- **len:** Etiqueta que identifica la variable.
- **equ:** Directiva que asigna un valor a una etiqueta.
- **$ - msg:** Expresión que calcula la longitud del mensaje restando la dirección de inicio del mensaje de la dirección actual del ensamblador.

Informe de Investigación

Referencias Bibliográficas

- https://www.tutorialspoint.com//assembly_programming/index.htm
- https://owasp.org/www-project-top-10-for-large-language-model-applications/
- https://asm86.wordpress.com/2009/10/12/ensamblador-desde-cero/
- https://www.djangoproject.com/weblog/2024/dec/04/security-releases/

Informe de Investigación

Elaborado por:

Especialista. Joelymar Uzcategui

DE USO PÚBLICO