Сем2

Сегодня 23:11 Без категории

Функция	EAX	EDX	ECX	EFLAGS
io_get_dec io_get_udec io_get_hex	выход: число			
io_get_char	выход: символ			
io_get_string	вход: адрес буфера	вход: размер строки		
io_print_dec io_print_udec io_print_hex	вход: число			
io_print_char	вход: символ			
io_print_string	вход: адрес буфера			
io_newline				

Таблица 1: Сводная таблица функций ввода/вывода

* Пример sum.s, hello_world.s; Обратить внимание на возвращаемое значение.

* У переменных в ассемблере нет типов – это просто именованные участки памяти. Корректная интерпретация данных лежит на программисте.

* .data - данные с инициализацией

data_init.s –

- var1 : dw - объявляем переменную размера word (16 бит, 2 байта) -
  - спросить, помнят ли с лекции
- var2 : dd - 32 бита, double word

- db - 1 байт (byte)
- dw - 2 байта (word)
- dd - 4 байта (double word)

- var1 - адрес
  [var1] - дальше по этому адресу

- Явное указание размера операнда: BYTE, WORD, DWORD & QWORD
- Если не указывать размер явно, то определяется по размеру операнда

Data_init.S последний оператор, пример автора.

Пример логического определения порядка, почему такое странное число - ответ в тесте про little-endian

Пример bss_init.S

- bss - Секция памяти для хранения неинициализированных данных.
- Т.е. мы не хотим задать значение, но хотим зарезервировать память.
  Она будет обнулена
- Директивы резервирования памяти
  - resb - 1 байт x size
  - resw - 2 байта x size
  - resd - 4 байта x size
- size указываем после директивы

Константы:

Пример const.S

- Директива equ позволяет определять константы, которые будут заменяться ассемблером при компиляции
- Константы можно использовать при операциях

Пример little_endian

- little-endian; big endian_ порядок хранения байтов в памяти

Пусть, есть некоторые данные:

0x12345678

- Big - endian

Addr	0x100	0x101	0x102	0x103
Byte	0x12	0x34	0x56	0x78

- Little - endian

Addr	0x100	0x101	0x102	0x103
Byte	0x78	0x56	0x34	0x12

- Почему little-endian?

- если файлы - байты

Пусть это массив char:

- Тогда младший байт по младшему адресу

- Пусть p - указатель на полное число 0x12345678

У p тип uint32_t

- Тогда: *(uint8_t)p => 0x78

*(uint16_t)p => 0x5678

- Все работает естественно:
  - uint8_t - младшая половина от uint16_t
  - uint16_t - младшая половина от uint32_t

- Пример с ASCII

- сначала пытается строка
- потом пытается адрес перешитой в виртуальном адресном пространстве

– Сделать objdump, найти Hello world'а, показать совпадение адресов, сравнить по ASCII

– Пример mov.s + objdump

1) Загрузка адреса byte_var – видим на экране 32-битное число

2) Загрузка значения byte_var. Т.к. не указали размер области для загрузки – грузит 32 бита (размер eax).
• byte_var размером 8 бит => видим лишний мусор (на самом деле часть word_var

3) Zero_extended загрузка eax – 32 бита; byte_var – 8 бит остальное дополняется нулями это делает инструкция movzx

4) Sign_extended

- byte_var = 0x FF, т.е. по сути byte_var = -1
- Мы делаем sign-extended mov: movsx
- Она дополняет старшие биты целевого регистра копией знакового бита

5) Чтение размером WORD

6) Чтение размером DWORD

7) Запись в память:
- Всегда нужно указывать размер т.к. тебе нельзя определить из destination операнда

8) Аналогично первому:
- Не указали размер => => считаем 32 бита
- Но word_var - 16 бит => видим мусор. (содержимое

bword = war)

9) Аналогично и по:
- По сути мы вышли за пределы секции .data
- Мы вышли в выводе правильное число, но это везение.
- Мы залезли в .bss, которая инициализирована нулями

Пример Corruption.s

- Мы залезли в другую переменную

=> Вывод: никогда под влиянием, потому что активно такое быть не должно.