汇编语言与逆向技术

第6章 数据访问

王志
zwang@nankai.edu.cn

南开大学 网络空间安全学院

2022-2023学年

本章学习的知识点

1. 数据传送指令
2. 加法和减法
3. 数据相关操作符和伪指令
4. 间接寻址

1. 数据传送指令

汇编语言与高级语言

- 汇编语言与高级语言最根本的不同之处在于，程序员必须掌握内存中的数据存储和机器与系统相关的大量细节
- 汇编语言给了程序员极大的自由，可以直接与机器对话，不需要依靠各种“翻译人员”

操作数类型

- 立即数（immediate）
  - mov eax, 10h
- 寄存器（register）
  - inc eax
- 内存（memory）
  - mov eax, [ebp+8]

寄存器操作数的简写符号

- r8，8位通用寄存器
- r16，16位通用寄存器
- r32，32位通用寄存器
- reg，任意的通用寄存器
- sreg，16位段寄存器

立即数操作数的简写符号

- imm，8位、16位或32位立即数
- imm8， 8位立即数
- imm16，16位立即数
- imm32，32位立即数

内存操作数的简写符号

- r/m8，8位通用寄存器或内存操作数
- r/m16，16位通用寄存器或内存操作数
- r/m32，32位通用寄存器或内存操作数
- mem，8位、16位、32位内存操作数

MOV指令

- mov指令从源操作数向目的操作数复制数据
  - mov destination，source
  - C++中，destination = source

MOV指令

- 两个操作数的尺寸必须一致
- 两个操作数不能同时为内存操作数
- 目的操作数不能是CS、EIP和IP
- 立即数不能直接送至段寄存器

mov指令

- **mov** — Move (Opcodes: 88, 89, 8A, 8B, 8C, 8E, ...)
- 语法
  - mov <reg>,<reg>
  - mov <reg>,<mem>
  - mov <mem>,<reg>
  - mov <reg>,<imm>
  - mov <mem>,<imm>
- 例子
  - mov byte ptr [var], 5

直接内存操作数

```
.data
    var1 dword 1000h ； 内存位置偏移00403000h
.code
    mov eax, var1 ； 机器指令 A100304000
```

- 变量名（数据标号）
  - 数据段内偏移地址

内存寻址操作

- masm32使用方括号表示内存寻址操作
  - mov eax, [var1]
- 通常，直接内存操作数不使用中括号
  - mov eax, var1
- 涉及到算术表达式时，使用中括号
  - mov eax，[var1+5]

invalid instruction operands

```
.data
    var1 DWORD 1000h
    var2 DWORD 2000h

.code

start:
    mov eax, var1
    mov var1, var2
    invoke ExitProcess, 0

end start
```

```
D:\>\masm32\bin\ml /c /coff hello.asm
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997.  All rights reserve

 Assembling: hello.asm

**********
ASCII build
**********

hello.asm(29) : error A2070: invalid instruction operands
```

内存之间的数据移动

```
.data
    var1 DWORD 0
    var2 DWORD 100h
.code
    mov eax, var2
    mov var1, eax
```

整数的零扩展

- 复制尺寸较小的操作数到尺寸较大的操作数
- MOVZX指令 (move with zero-extend)
  - movzx r32, r/m8
  - movzx r32, r/m16
  - movzx r16, r/m8

MOVSX

- MOVSX（move with sign-extend）符号扩展传送指令，最高位循环填充所有扩展位
  - 有符号整数的存储空间扩展
  - movsx r32，r/m8
  - movsx r32，r/m16
  - movsx r16，r/m8

LAHF指令

- LAHF（load status flags into AH）指令把EFLAGS寄存器的低字节复制到AH寄存器
  - 符号标志（SF）
  - 零标志(ZF)
  - 辅助进位标志(AF)
  - 奇偶标志(PF)
  - 进位标志(CF)

SAHF指令

- SAHF（store AH into status flags）指令复制AH寄存器的值至EFLAGS寄存器的低字节
  - 修改CPU的符号标志（SF）、零标志(ZF)、辅助进位标志(AF)、奇偶标志(PF)、进位标志(CF)

XCHG指令

- XCHG（exchange data）指令交换两个操作数的内容
  - XCHG reg, reg
  - XCHG reg, mem
  - XCHG mem, reg

交换两个内存的值

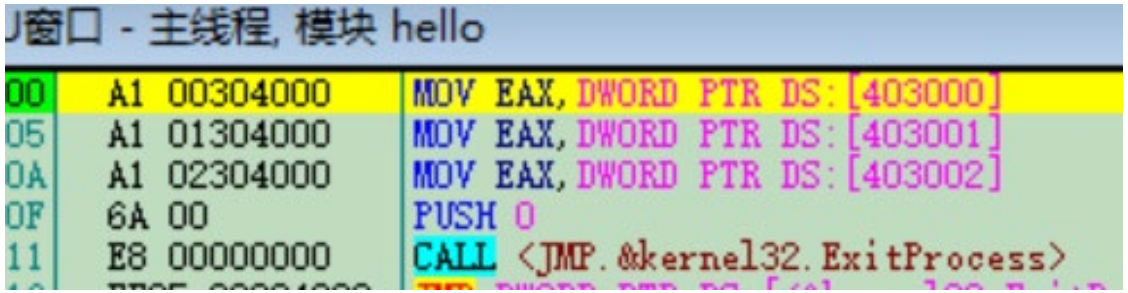
```
.data
    var1 DWORD 100h
    var2 DWORD 200h
.code
    mov eax, var1
    xchg eax, var2
    mov var1, eax
```

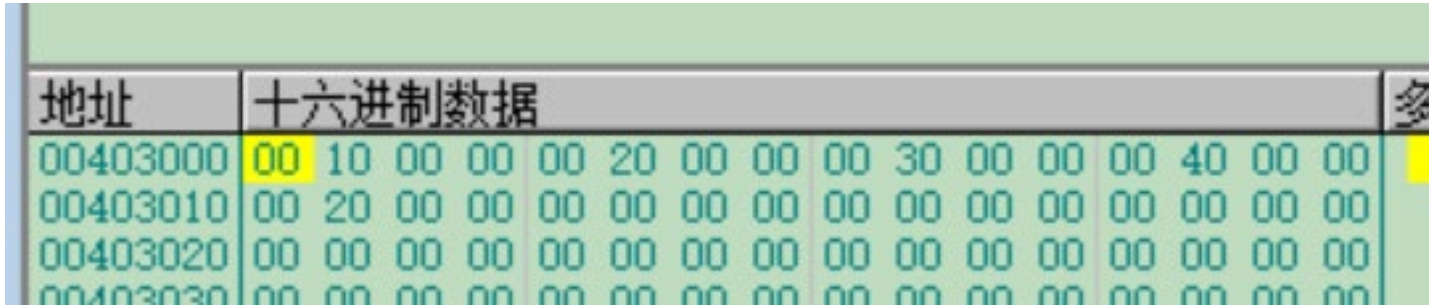
直接偏移操作数

- 在变量名后面加上一个偏移值，可以创建直接偏移（direct-offset）操作数
- 访问没有显式标号的内存

直接偏移操作数

```
.data
    var1 DWORD 1000h, 2000h, 3000h, 4000h
.code
start:
    mov eax, var1
    mov eax, [var1+1]
    mov eax, [var1+2]
    invoke ExitProcess, 0
```





2. 加法和减法

INC指令

- INC（increment）指令从操作数中加1
- 语法
  - inc <reg>

    inc <mem>
- 例子
  - inc eax
  - inc [var1+4]

DEC指令

- DEC（decrement）指令从操作数中减1
- 语法

  dec <reg>

  dec <mem>

- 例子

  inc eax

  inc [var1+4]

ADD指令

- ADD指令将同尺寸的源操作数和目的操作数相加

  add <reg>,<reg>

  add <reg>,<mem>

  add <mem>,<reg>

  add <reg>,<imm>

  add <mem>,<imm>

- 相加的结果存储在目的操作数中
  - add 目的操作数，源操作数
  - add eax, ebx ---- eax ← eax + ebx
  - 影响标志位CF、ZF、SF、OF、AF、PF

SUB指令

- SUB指令将源操作数从目的操作数中减掉

  sub <reg>,<reg>

  sub <reg>,<mem>

  sub <mem>,<reg>

  sub <reg>,<imm>

  sub <mem>,<imm>

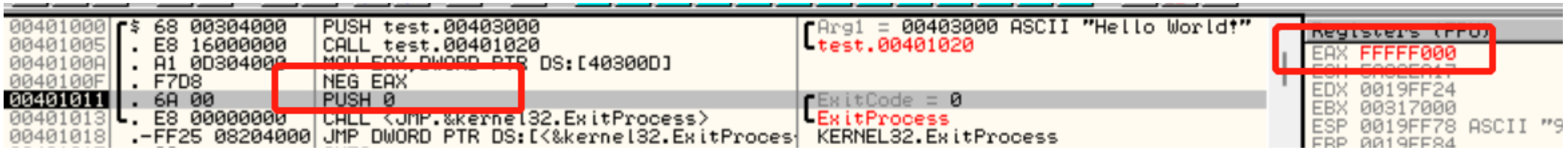
- SUB 目的操作数， 源操作数
  - sub eax, ebx --- eax ← eax - ebx
- 影响的标志位有CF、ZF、SF、OF、AF、PF

NEG指令

- NEG（negate）指令通过将数字转换为对应的补码而求得其相反数
  - neg <reg>

    neg <mem>
  - 相当于乘以-1。正数变成负数，负数变成正数
  - 1→ 0001, -1→1111
- 影响的标志位：CF、ZF、SF、OF、AF、PF


00401000 $ 68 00304000 PUSH test.00403000
00401005 . E8 16000000 CALL test.00401020
0040100A . A1 0D304000 MOV EAX,DWORD PTR DS:[40300D]
0040100F . F7D8 NEG EAX
00401011 . 6A 00 PUSH 0
00401013 . E8 00000000 CALL <JMP.&kernel32.ExitProcess>
00401018 .-FF25 08204000 JMP DWORD PTR DS:[<&kernel32.ExitProces
Arg1 = 00403000 ASCII "Hello World!"
test.00401020
ExitCode = 0
ExitProcess
KERNEL32.ExitProcess
Registers (FPU
EAX 00001000
EDX 0019FF24
EBX 00317000
ESP 0019FF78 ASCII "9gWv"


00401000 $ 68 00304000 PUSH test.00403000
00401005 . E8 16000000 CALL test.00401020
0040100A . A1 0D304000 MOV EAX,DWORD PTR DS:[40300D]
0040100F . F7D8 NEG EAX
00401011 . 6A 00 PUSH 0
00401013 . E8 00000000 CALL <JMP.&kernel32.ExitProcess>
00401018 .-FF25 08204000 JMP DWORD PTR DS:[<&kernel32.ExitProces
Arg1 = 00403000 ASCII "Hello World!"
test.00401020
ExitCode = 0
ExitProcess
KERNEL32.ExitProcess
EAX FFFFF000
EDX 0019FF24
EBX 00317000
ESP 0019FF78 ASCII "9

3. 数据相关操作符和伪指令

数据相关的伪指令

- BYTE、WORD、DWORD
- ALIGN伪指令
- LABEL伪指令

Directives Reference

08/04/2021 • 2 minutes to read • 

Data Allocation

ALIGN	FWORD	REAL8
BYTE	LABEL	REAL10
SBYTE	ORG	TBYTE
DWORD	QWORD	WORD
SDWORD	REAL4	SWORD
EVEN		

数据相关的操作符（Operator）

- PTR操作符
- TYPE操作符
- LENGTHOF操作符
- SIZEOF操作符
- OFFSET操作符

MASM Operators reference

08/04/2021 • 2 minutes to read • +3

Type

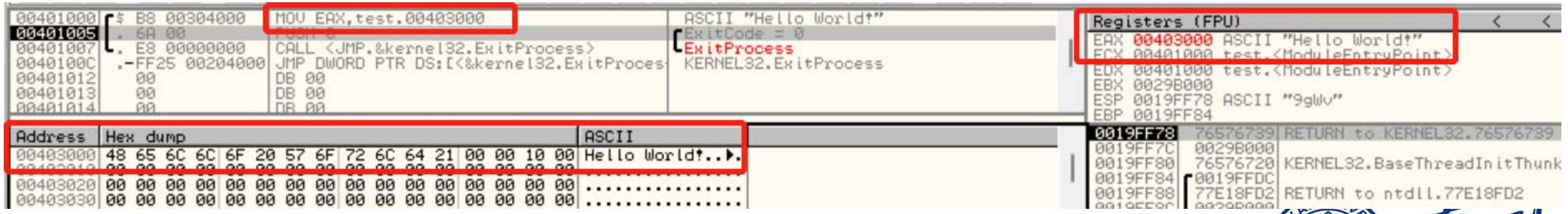
HIGH (high 8 bits of lowest 16 bits)
HIGH32 (high 32 bits of 64 bits)
HIGHWORD (high 16 bits of lowest 32 bits)
LENGTH (number of elements in array)
LENGTHOF (number of elements in array)
LOW (low 8 bits)
LOW32 (low 32 bits)
LOWWORD (low 16 bits)
OPATTR (get argument type info)
PTR (pointer to or as type)
SHORT (mark short label type)
SIZE (size of type or variable)
SIZEOF (size of type or variable)
THIS (current location)
TYPE (get expression type)
.TYPE (get argument type info)

OFFSET操作符

- OFFSET操作符返回数据标号的偏移地址
- 偏移地址表示标号距离数据段开始的距离
  - CS的值一般是0
  - CS为零的时候，OFFSET等同内存虚拟地址

```
.data
str_hello db "Hello World!", 0
var dd 1000h


.code
start:
mov eax, offset str_hello
invoke ExitProcess, 0
END start
```

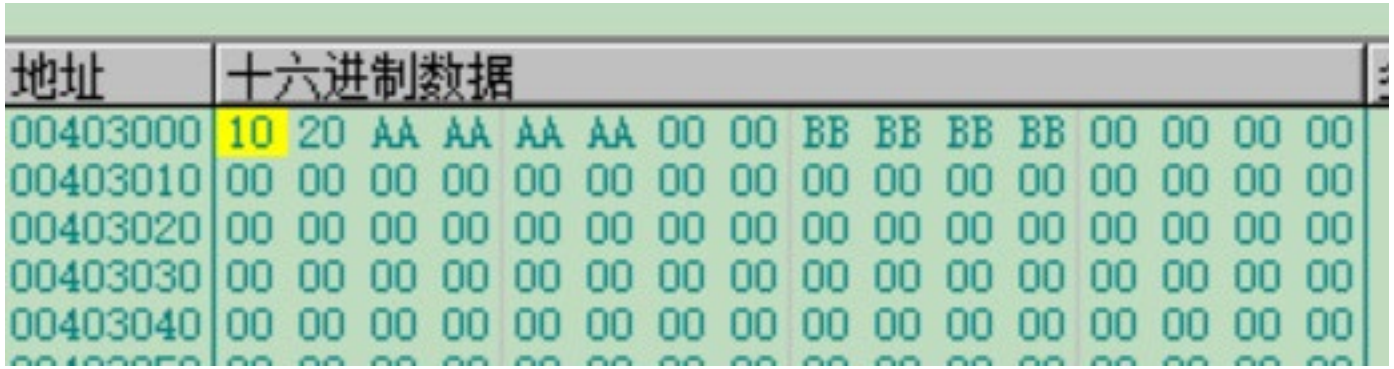


ALIGN伪指令

- ALIGN指令将变量的位置按BYTE、WORD、DWORD边界对齐
  - ALIGN 边界值
  - 边界值可以是1、2、4、8或16（ a power of 2 ）
  - “Aligned data can improve performance, at the expense of wasted space between data elements.”

ALIGN伪指令

```
.data
    var1 BYTE 10h, 20h
    var2 DWORD 0AAAAAAAAh
    ALIGN 4
    var3 DWORD 0BBBBBBBBh
```



地址	十六进制数据															
00403000	10	20	AA	AA	AA	AA	00	00	BB	BB	BB	BB	00	00	00	00
00403010	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00403020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00403030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00403040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

ALIGN

- When data is aligned, the skipped space is padded with zeroes. When instructions are aligned, the skipped space is filled with appropriately-sized NOP instructions.
- https://docs.microsoft.com/en-us/cpp/assembler/masm/align-masm?view=msvc-160

PTR操作符

- PTR操作符可以重载操作数声明的默认尺寸

```
.data
    var1 DWORD 12345678h

.code
start:
    movzx eax, BYTE PTR var1
    movzx ebx, BYTE PTR [var1+1]
    invoke ExitProcess, 0
```

TYPE操作符

- TYPE操作符返回变量的字节数

```
.data
    var1 db 0
    var2 dw 0
    var3 dd 0
.code
    mov eax, type var2
```

LENGTHOF操作符

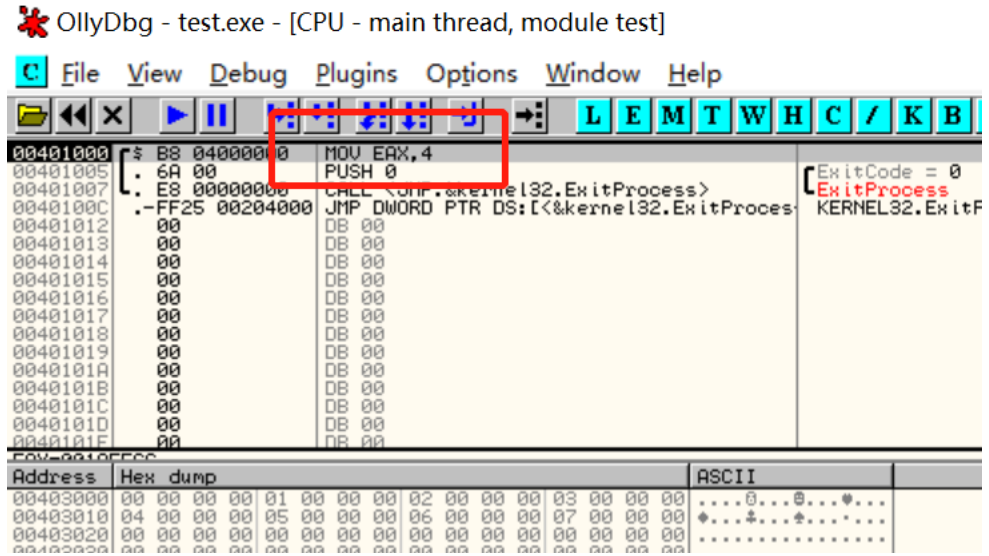
- LENGTHOF操作符计算数组中元素的数目，元素由出现在同一行的值定义

```
.data
    var1 DWORD   0，1，2，3
.code
    mov eax, LENGTHOF var1
```

LENGTHOF操作符

```
.data
var1 dd   0, 1, 2, 3
      dd   4, 5, 6, 7

.code
start:
mov eax, lengthof var1
invoke ExitProcess, 0
end start
```



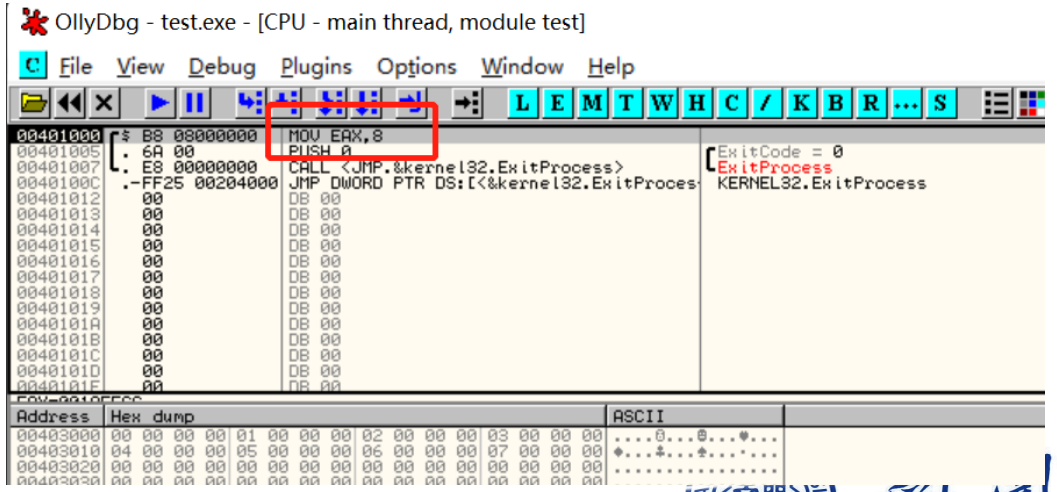
LENGTHOF操作符

```
.data
    var1 DWORD  0，  1，  2，3,
                4，  5，  6，7
.code
    mov eax, LENGTHOF var1
```

- 第一行的最后加一个逗号，连接下一行的初始值

```
.data
var1 dd   0, 1, 2, 3,
          4, 5, 6, 7

.code
start:
mov eax, lengthof var1
invoke ExitProcess, 0
end start
```



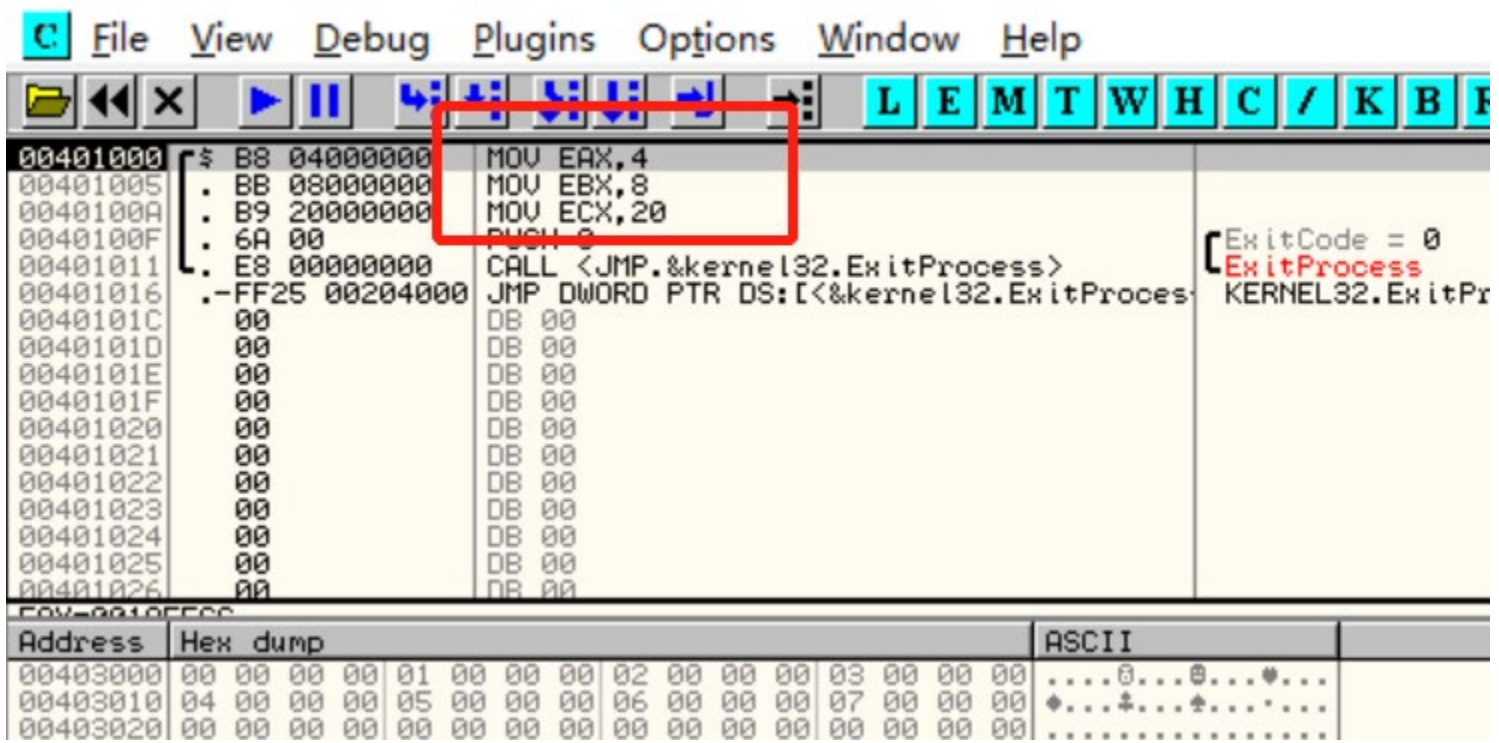
SIZEOF操作符

- SIZEOF操作符的返回值等于LENGTHOF和TYPE返回值的乘积

```
.data
    var1 DWORD  0，  1，  2，3,
                4，  5，  6，7
.code
    mov eax, SIZEOF var1
```

```
.data
var1 dd   0, 1, 2, 3,
          4, 5, 6, 7

.code
start:
mov eax, type var1
mov ebx, lengthof var1
mov ecx, sizeof var1
invoke ExitProcess, 0
end start
```



LABEL伪指令

- LABEL伪指令允许插入一个标号，并赋予其尺寸属性而无须分配任何实际的存储空间。
- 为数据段内其后定义的变量提供一个别名

LABEL伪指令

```
.data
    dd_var label dword
    dw_var1 dw 1234h
    dw_var2 dw 5678h
.code
    mov eax, dd_var
```

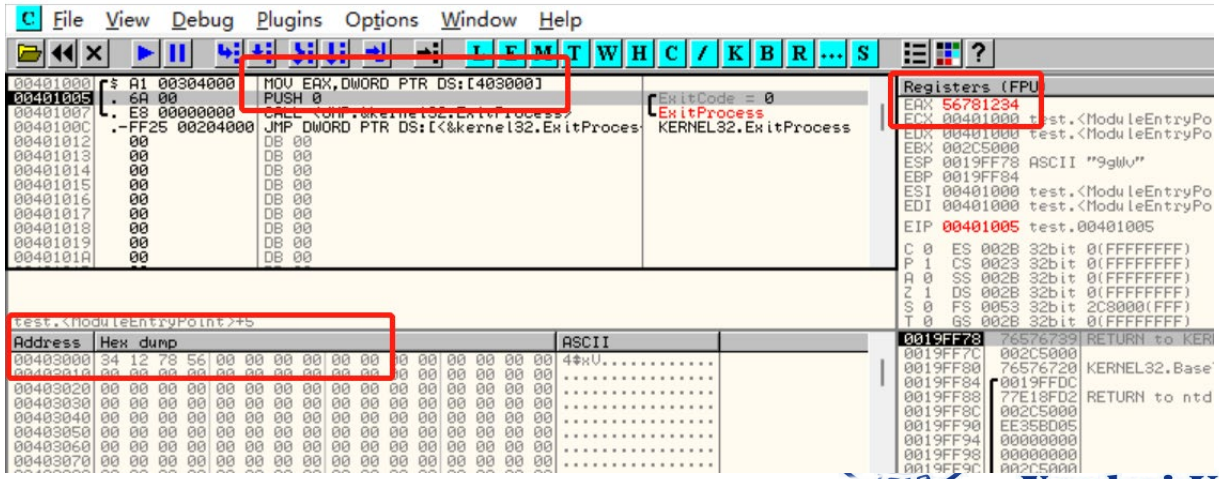
eax等于56781234h

```
.data
dd_var label dword
dw_var1 dw 1234h
dw_var2 dw 5678h

.code
start:
mov eax, dd_var
invoke ExitProcess, 0
end start
```

注意：label后面的数据类型不能用缩写，例如dword不能写成dd

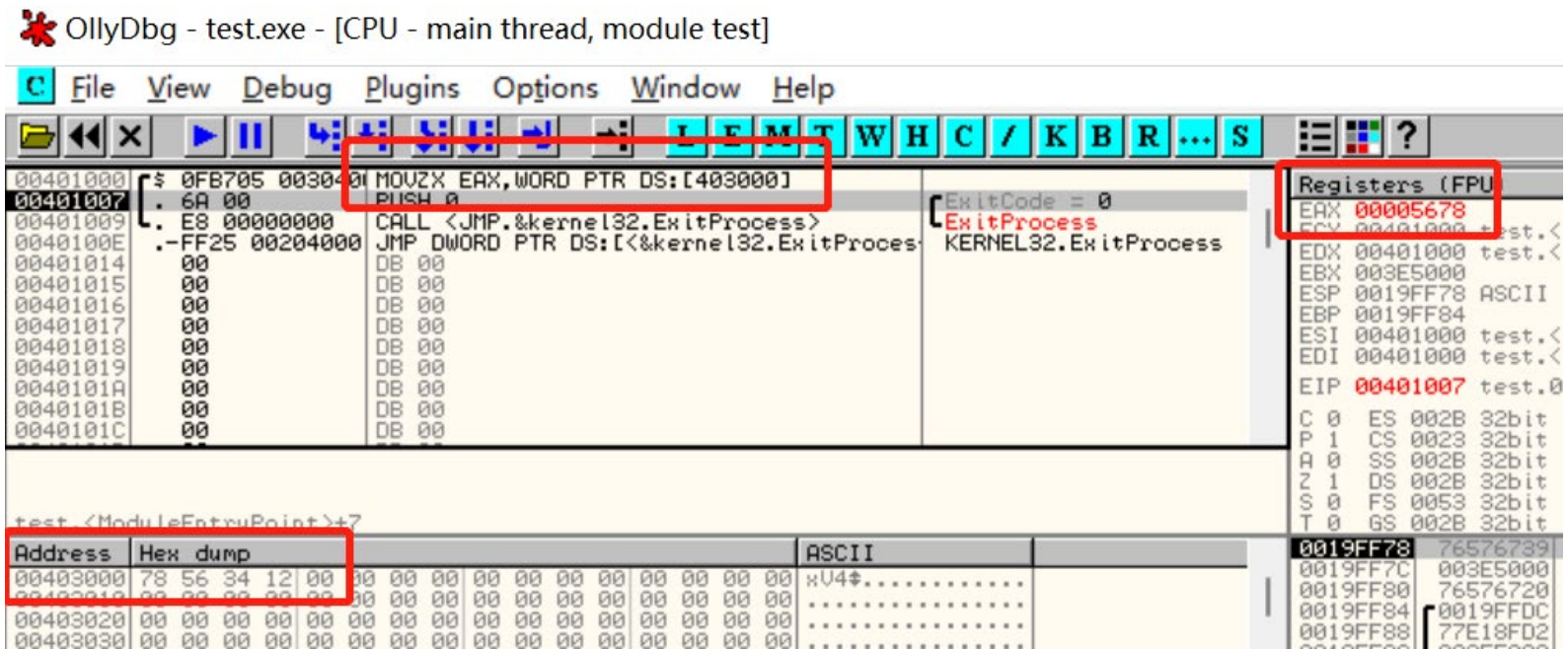
```
test.asm(12) : error A2008: syntax error : dd
test.asm(18) : error A2006: undefined symbol : dd_var
```



```
.data

dw_var label word
dd_var dd 12345678h

.code
start:
movzx eax, dw_var
invoke ExitProcess, 0
end start
```



4. 间接寻址

间接寻址

- 用寄存器作为指针并控制该寄存器的值称为间接寻址（indirect addressing）
- 如果一个操作数使用的是间接寻址，就称之为间接操作数（indirect operand）。

间接操作数

- 任何一个 32 位通用寄存器（EAX、EBX、ECX、EDX、ESI、EDI、EBP 和 ESP）加上方括号就能构成一个间接操作数

```
.data

dw_var label word
dd_var dd 12345678h

.code
start:
mov eax, offset dd_var
inc [eax]
invoke ExitProcess, 0
end start
```

```
************
ASCII build
************

test.asm(18) : error A2023: instruction operand must have size
PS D:\>
```

间接操作数

```
.data
    val  dd 12345678h
.code
    mov esi, offset val
    mov eax, dword ptr [esi]
```

间接操作数

```
.data
array_dw DWORD 10000h, 20000h, 30000h
.code
mov esi, offset array_dw
mov eax, [esi] ;（第一个数）
add esi, 4
add eax, [esi] ;（第二个数）
add esi, 4
add eax, [esi] ;（第三个数）
```

变址操作数

- 变址操作数（indexed operand）把常量和寄存器相加以得到一个有效地址
- 任何32位通用寄存器都可以作为变址寄存器
  - constant[*reg*]
  - [constant+*reg*]

变址操作数

```
.data
array_dw DWORD 10000h, 20000h, 30000h
.code
mov esi, 0
mov eax, array_dw[esi] ;（第一个数）
add esi, 4
add eax, array_dw[esi] ;（第二个数）
add esi, 4
add eax, array_dw[esi] ;（第三个数）
```

变址操作数

```
.data
array_dw DWORD 10000h, 20000h, 30000h
.code
mov esi, OFFSET array_dw
mov eax,  [esi] ;（第一个数）
add eax, [esi+4] ;（第二个数）
add eax, [esi+8] ;（第三个数）
```

变址操作数的比例因子

```
.data
array_dw DWORD 10000h, 20000h, 30000h
.code
mov esi, 0
mov eax, array_dw[esi*TYPE array_dw] ;
mov esi, 1
add eax, array_dw[esi* TYPE array_dw]
mov esi, 2
add eax, array_dw[esi* TYPE array_dw]
```

基址变址操作数（base-indexed）

- 基址变址操作数把两个寄存器的值相加，得到一个偏移地址
  - [base + index]
  - mov eax, [ebx + esi]
- 基址寄存器和变址寄存器可以使用任意的32位通用寄存器

相对基址变址操作数

- 相对基址变址（based-indexed with displacement）操作数把偏移、基址、变址以及可选的比例因子组合起来，产生一个偏移地址。
  - [base + index + displacement]
  - displacement[base + index]

相对基址变址操作数

```
.data
table  dd 10000h, 20000h, 30000h
row_size = ($ - table)
        dd 40000h, 50000h, 60000h
        dd 70000h, 80000h, 90000h
.code
mov ebx, row_size
mov esi, 2
mov eax, table[ebx + esi * type table]
```

指针

- 如果一个变量包含另一个变量的地址，则该变量称为指针

指针

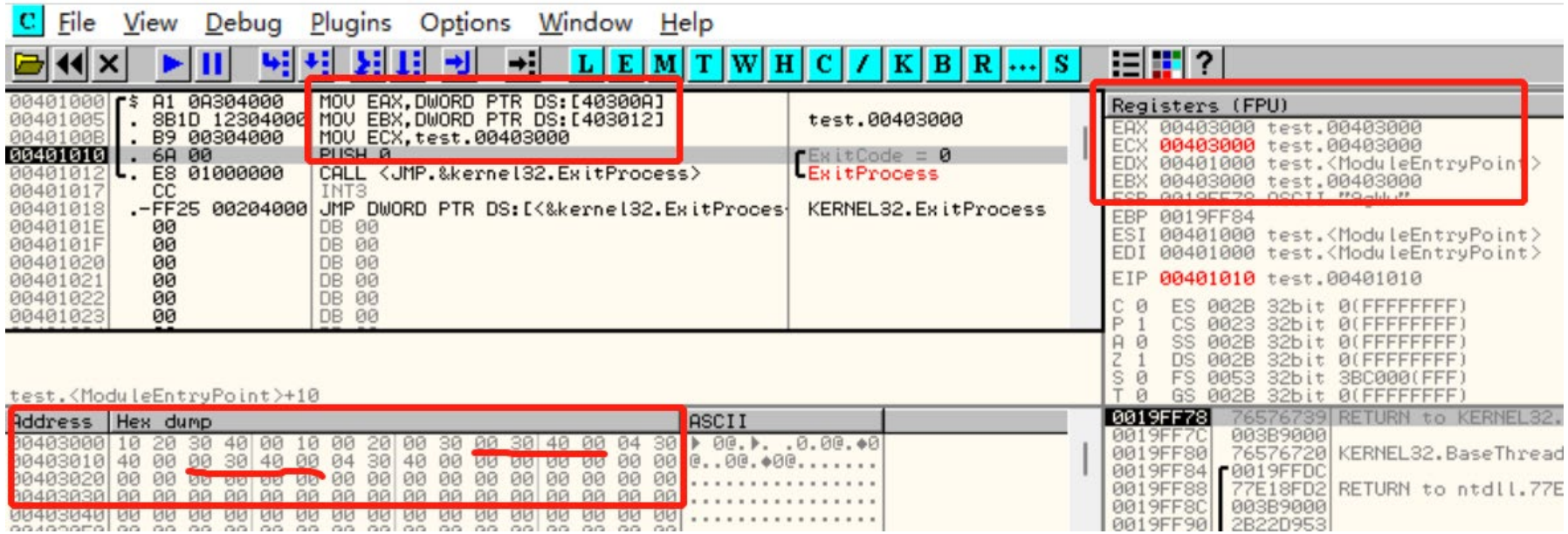
```
.data
array_b   db  10h, 20h, 30h, 40h
array_w  dw 1000h, 2000h, 3000h
ptr_b   dd  array_b
ptr_w  dd  array_w
```

指针

```
.data
array_b  db 10h, 20h, 30h, 40h
array_w  dw 1000h, 2000h, 3000h
ptr_b  dd offset array_b
ptr_w  dd offset array_w
```

```
array_b  db 10h, 20h, 30h, 40h
array_w  dw 1000h, 2000h, 3000h
ptr_b  dword offset array_b
ptr_w  dword offset array_w
ptr_b1   dd  array_b
ptr_w1  dd  array_w

.code
start:
mov eax, ptr_b
mov ebx, ptr_b1
mov ecx, offset array_b
```



指针

- 32位模式下的NEAR指针和FAR指针
- **NEAR指针（课程使用NEAR指针）**
  - 相对数据段开始的32位偏移地址
- FAR指针
  - 48位的段选择子-偏移地址

TYPEDEF操作符

- TYPEDEF操作符允许创建用户自定义的类型
  - PBYTE TYPEDEF PTR BYTE ;字节指针
  - PWORD TYPEDEF PTR WORD ;字指针
  - PDWORD TYPEDEF PTR DWORD ;双字指针

TYPEDEF操作符

```
PADWORD TYPEDEF PTR DWORD

.data

        array1 dd 1000h, 2000h, 3000h, 4000h

        ptr1 PADWORD array1
```

本章学习的知识点

1. 数据传送指令
2. 加法和减法
3. 数据相关操作符和伪指令
4. 间接寻址

汇编语言与逆向技术

第6章 数据访问

王志
zwang@nankai.edu.cn

南开大学 网络空间安全学院

2022-2023学年