



汇编语言与逆向工程

北京邮电大学
2019年3月



第二章 基础知识

- 1. 寄存器
- 2. x86指令集
- 3. 寻址方式
- 4. 字节序
- 5. 栈
- 6. 函数调用约定



5.栈

@ 栈

□ 一种线性的数据结构

○ 先入后出，

❖ 后入栈的数据会比之前入栈的数据先出栈。

○ 栈操作主要有两种——压栈和弹栈，

❖ 每次压栈会把数据从栈顶压入，栈的长度增加，栈顶位置上升；

❖ 每次弹栈也是从栈顶弹出一个数据，栈的长度减少，栈顶位置下降。

○ 依次向栈中压入1、2、3三个数据，其弹出顺序为3、2、1。



5.栈

- ④ 程序栈
- ④ 栈的布局
- ④ 栈的生成与销毁



5.栈

@程序栈

- 栈在程序中起着不可替代的作用
 - 每一个函数拥有自己的函数栈
 - 函数栈中保存着函数的局部变量、流控制结构等等
 - 每次一个函数调用后函数栈会被收回，并再次分配给其他被调用的函数
- 程序栈是由高地址向低地址生长
 - 在程序中函数站的栈顶地址是比栈底地址低的



5.栈

@ 程序栈

- **x86**框架为例，支持栈的寄存器包括**ESP**和**EBP**，以**ESP**和**EBP**作为栈的边界
- **ESP**是栈寄存器，指示当前栈顶的位置
 - ❖ 该寄存器在一个函数调用过程中会随着数据的压入弹出不断改变
- **EBP**是栈基址寄存器，这个寄存器指向栈底的位置
 - ❖ 主要作用是做栈划分
 - ❖ 在一个函数执行的过程中，该寄存器的值在调用其他函数时暂时改变为其他函数栈的栈底位置，当其他函数调用结束后会恢复为原函数栈栈底位置



5.栈

@程序栈

- ❑ x86指令集中，与栈操作相关的指令：**push、pop、call、leave、ret**
 - push和pop是栈的基本操作
 - call、leave、ret可以用来构建函数栈和销毁函数栈。



低地址

当前函数栈

调用者栈

高地址



栈增长方向

← 栈寄存器 esp

局部变量等

保留调用者函数栈基址ebp

← 栈基址寄存器 ebp

返回地址

参数1

...

参数n



5.栈

@ 栈的布局

- ❑ 栈的生长是从高地址向低地址生长的，下图展示了栈在内存中是如何布局的。
- ❑ 每一次函数的调用会生成一个新的栈帧，栈帧可供函数存储局部变量，或调用其他函数。
- ❑ 当函数执行结束后，该函数栈帧会被释放，**ESP**、**EBP**会重新赋值为上一栈帧栈顶、栈底位置。
- ❑ 栈帧在内存中会不断的分配、释放



5.栈

@ 栈的生成与销毁

- ❑ 从一个简单的程序，从汇编语言层面观察栈帧的生成与销毁
- ❑ 使用**ida pro**工具对编译好的可执行文件进行反汇编
- ❑ 看一下关于**main**函数和**foo**函数的汇编代码



5.栈

```
#include<stdio.h>
Int foo(int a){
    printf("argv = %d",a);
    return 0;
}
Int main(){
    int a;
    printf("hello stack \n");
    a=1;
    foo(a);
    return 0;
}
```



子函数调用过程

- 当函数调用发生时，新的栈帧被压入栈中，而当函数返回时，栈帧将从栈中弹出。

④ 第一步：Push 参数1 参数入栈。

- ❑ 母函数调用子函数时，在母函数栈帧再往内存低地址方向的邻接区域，创建子函数的栈帧，首先把参数压入栈中；

④ 第二步：call 子函数地址

- ❑ 返回地址入栈，将指令寄存器eip中保存的下一条执行指令的地址做为返回母函数的返回地址压入栈，当子函数调用结束后返回时，程序应该按照返回地址跳转到母函数的下一条指令继续执行；
- ❑ 代码区跳转，处理器从当前母函数的代码区跳转到子函数程序的入口，即程序的控制权转移到被调用的子函数；



低地址

当前函数栈

调用者栈

高地址



栈增长方向

← 栈寄存器 esp

← 栈基址寄存器 ebp

局部变量等

保留调用者函数栈基址ebp

返回地址

参数1

...

参数n



子函数调用过程

④ 第三步：push ebp ；

- ❑ ebp中母函数栈帧基址指针入栈，子函数将基址寄存器ebp中保存的母函数栈帧的基地址指针压入栈中保存；

④ 第四步：mov ebp esp ；

- ❑ esp值装入ebp，ebp更新为新栈帧基地址。把esp中保存的最新栈顶指针拷贝到基址寄存器ebp中，这时ebp中保存的就是正在被调用子函数的基地址,即子函数的栈帧底部地址；

④ 第五步：sub esp xxx ；

- ❑ 给新栈帧分配空间，根据函数需要保存局部变量的空间大小，栈顶指针esp从子函数的基地址向内存低地址偏移，为局部变量留出一定空间



低地址

当前函数栈

调用者栈

高地址



栈增长方向

← 栈寄存器 esp

← 栈基址寄存器 ebp

局部变量等

保留调用者函数栈基址ebp

返回地址

参数1

...

参数n



@第六步: **mov esp, ebp**

- ❑ 栈顶指针**esp**恢复到栈底位置，也就是将分配的栈空间全部释放，将分配的栈空间全部释放

@第七步: **pop ebp**

- ❑ 将**ESP**会指向的栈帧中母函数栈帧基址指针弹出保存在**ebp**

@第八步: **ret**

- ❑ 按照返回地址指向的位置，返回**main**函数下一步指令



5.栈

□foo函数的调用步骤如下:

- 1.在.text 40109c处, 执行了mov eax, [ebp+var_4]指令, ebp+var_4位置的偏移是变量a在栈帧中的位置, 继续执行了push eax指令, 导致a被压入栈中, ESP向低地址增长4字节。
- 2.在.text 4010a0执行了call foo指令, call指令会将调用函数结束的返回地址, 即函数中下一条指令add esp,4的地址0x4010a5压入栈中, 程序进入foo函数中执行。



5.栈

④ **foo**函数的调用步骤如下：

- 3.进入**foo**函数后，首先执行的是**push ebp**指令，该指令的目的是保存上一函数栈的栈基址位置，便于函数结束后恢复上一函数栈。指令执行后，**EBP**中的数据被压入栈中，**ESP**向低地址增长**4**字节。
- 4.在**.text 401021**执行了**mov ebp, esp**指令，该指令将**EBP**寄存器的值改写为当前**ESP**寄存器的值，导致栈底上移，新栈开始分配。
- 5.在**.text 401023**执行了**sub esp, 40h**的指令，该指令均导致**ESP**寄存器向低地址增长了**64**字节，也就是说栈的长度为**64**字节，新栈分配到此结束。



5.栈

④ **foo**函数的调用步骤如下：

- 6.从.text 401023至.text 401053为函数调用了**printf**函数，是函数正常的执行逻辑及编译器添加的检查函数。
- 7.在.text 401058处开始执行了**mov esp, ebp**和**pop ebp**两条操作，在某些编译器中这两条也被整合成为**leave**指令。
 - ❖首先**mov esp, ebp**指令将栈顶指针**esp**恢复到栈底位置，也就是将分配的栈空间全部释放。
 - ❖再执行**pop ebp**指令，由之前的步骤可知，当前**ESP**寄存器指向空间的内容是**main**函数栈的栈基址位置，指令结束后，**ESP**会指向返回地址位置，**EBP**寄存器会恢复到**main**函数函数栈的栈底位置。



5.栈

④ **foo函数的调用步骤如下：**

- ④ 8.在.text 40105b处执行了**ret**指令，该指令内容可以理解为**pop eip**，执行后程序跳转回**main**函数，并且栈帧也恢复回到**main**函数执行**call foo**之前的状态。



5.栈

@ foo函数的调用步骤

- @ IDA的示例

- @ OllyDbg的动态调试过程



5.栈

@ foo函数的调用步骤如下:

```
.text:00401070 _main_0      proc near          ; CODE XREF: _main↑j
.text:00401070
.text:00401070 var_44      = byte ptr -44h
.text:00401070 var_4       = dword ptr -4
.text:00401070
.text:00401070      push    ebp
.text:00401071      mov     ebp, esp
.text:00401073      sub     esp, 44h
.text:00401076      push    ebx
.text:00401077      push    esi
.text:00401078      push    edi
.text:00401079      lea     edi, [ebp+var_44]
.text:0040107C      mov     ecx, 11h
.text:00401081      mov     eax, 0CCCCCCCCh
.text:00401086      rep stosd
.text:00401088      push    offset aHelloStack ; "hello stack \n"
.text:0040108D      call    _printf
.text:00401092      add     esp, 4
.text:00401095      mov     [ebp+var_4], 1
.text:0040109C      mov     eax, [ebp+var_4]
.text:0040109F      push    eax
.text:004010A0      call    sub_401005
.text:004010A5      add     esp, 4
.text:004010A8      xor     eax, eax
.text:004010AA      pop     edi
.text:004010AB      pop     esi
.text:004010AC      pop     ebx
.text:004010AD      add     esp, 44h
.text:004010B0      cmp     ebp, esp
.text:004010B2      call    __chkesp
.text:004010B7      mov     esp, ebp
.text:004010B9      pop     ebp
.text:004010BA      retn
.text:004010BA _main_0      endp
```

崔宝江



5.栈

@ foo函数的调用步骤如下:

```
.text:00401020 ; Attributes: bp-based frame
.text:00401020
.text:00401020 sub_401020      proc near                ; CODE XREF: sub_401005↑j
.text:00401020
.text:00401020 var_40          = byte ptr -40h
.text:00401020 arg_0           = dword ptr  8
.text:00401020
.text:00401020                push     ebp
.text:00401021                mov      ebp, esp
.text:00401023                sub      esp, 40h
.text:00401026                push     ebx
.text:00401027                push     esi
.text:00401028                push     edi
.text:00401029                lea      edi, [ebp+var_40]
.text:0040102C                mov      ecx, 10h
.text:00401031                mov      eax, 0CCCCCCCCh
.text:00401036                rep stosd
.text:00401038                mov      eax, [ebp+arg_0]
.text:0040103B                push     eax
.text:0040103C                push     offset aArgvD ; "argv = %d"
.text:00401041                call    _printf
.text:00401046                add      esp, 8
.text:00401049                xor      eax, eax
.text:0040104B                pop      edi
.text:0040104C                pop      esi
.text:0040104D                pop      ebx
.text:0040104E                add      esp, 40h
.text:00401051                cmp      ebp, esp
.text:00401053                call    __chkesp
.text:00401058                mov      esp, ebp
.text:0040105A                pop      ebp
.text:0040105B                retn
.text:0040105B sub_401020      endp
```



5.栈

④ **foo**函数的调用步骤如下:

④ OllyDbg的动态调试过程



第二章 基础知识

- 1. 寄存器
- 2. x86指令集
- 3. 寻址方式
- 4. 字节序
- 5. 栈
- 6. 函数调用约定



6.函数调用约定

@函数调用约定

- ❑ 对函数调用时如何传递参数的一种约定
 - 例如：使用VC++ 6.0对源代码进行编译，在函数定义时可以选择函数的调用约定
- ❑ 主要解决的是问题就是函数的参数是如何传递的，以及函数执行结束后，参数应该如何处理
- ❑ 主要的函数约定由三种
 - Cdecl
 - Stdcall
 - Fastcall



6.函数调用约定

```
#include<stdio.h>
void __cdecl foo_cdecl(int a,int b,int c){
    printf("argv %d %d %d \n",a,b,c);
}
void __stdcall foo_stdcall(int a,int b,int c){
    printf("argv %d %d %d \n",a,b,c);
}
void __fastcall foo_fastcall(int a,int b,int c,int d){
    printf("argv %d %d %d %d \n",a,b,c,d);
}
int main(){
    foo_cdecl(1,2,3);
    foo_stdcall(1,2,3);
    foo_fastcall(1,2,3,4);
    return 0;
}
```



6.函数调用约定

❑ **cdecl**是C/C++默认方式，参数从右向左入栈

```
.text:00401150      push     ebp
.text:00401151      mov      ebp, esp
.text:00401153      sub      esp, 40h
.text:00401156      push     ebx
.text:00401157      push     esi
.text:00401158      push     edi
.text:00401159      lea      edi, [ebp+var_40]
.text:0040115C      mov      ecx, 10h
.text:00401161      mov      eax, 0CCCCCCCCh
.text:00401166      rep stosl
.text:00401168      push     3
.text:0040116A      push     2
.text:0040116C      push     1
.text:0040116E      call     j__foo_cdecl
.text:00401173      add      esp, 0Ch
.text:00401176      push     3
.text:00401178      push     2
.text:0040117A      push     1
.text:0040117C      call     j__foo_stdcall@12 ; foo_stdcall(x,x,x)
.text:00401181      push     4
.text:00401183      push     3
.text:00401185      mov      edx, 2
.text:0040118A      mov      ecx, 1
.text:0040118F      call     j__foo_fastcall@12 ; foo_fastcall(x,x,x)
.text:00401194      xor      eax, eax
.text:00401196      pop      edi
.text:00401197      pop      esi
.text:00401198      pop      ebx
```

○ 查看main函数汇编代码



6.函数调用约定

□参数入栈顺序

- 参数是被压入栈中传递的，并且压入栈中的顺序是从右向左，即从最后一个参数开始压栈
- 此时相比未压栈时，**ESP**指针向减小了 $3*4=12$ （**0x0c**）。

□参数清除

- 当执行过**callfoo_cdecl**后，**ESP**指针是会恢复到压入参数后的位置，下一条执行**add esp, 0x0c**就会把**ESP**指针增加12字节，也就恢复到了压入参数之前的位置，从而实现了参数清除



6.函数调用约定

```
.text:00401030 arg_0      = dword ptr 8
.text:00401030 arg_4      = dword ptr 0Ch
.text:00401030 arg_8      = dword ptr 10h
.text:00401030
.text:00401030 push     ebp
.text:00401031 mov      ebp, esp
.text:00401033 sub      esp, 40h
.text:00401036 push     ebx
.text:00401037 push     esi
.text:00401038 push     edi
.text:00401039 lea      edi, [ebp+var_40]
.text:0040103C mov      ecx, 10h
.text:00401041 mov      eax, 0CCCCCCCCh
.text:00401046 rep stosd
.text:00401048 mov      eax, [ebp+arg_8]
.text:0040104B push     eax
.text:0040104C mov      ecx, [ebp+arg_4]
.text:0040104F push     ecx
.text:00401050 mov      edx, [ebp+arg_0]
.text:00401053 push     edx
.text:00401054 push     offset Format ; "argu %d %d %d \n"
.text:00401059 call     _printf
.text:0040105E add      esp, 10h
.text:00401061 pop      edi
.text:00401062 pop      esi
.text:00401063 pop      ebx
.text:00401064 add      esp, 40h
.text:00401067 cmp      ebp, esp
.text:00401069 call     __chkesp
.text:0040106E mov      esp, ebp
.text:00401070 pop      ebp
.text:00401071 retn
.text:00401071 _foo_cdecl endp
```

foo_cdecl反汇编代码中



6.函数调用约定

- ❑ 在foo_cdecl反汇编代码中，可以看到函数是利用的是**EBP**寄存器与参数位置的相对关系顺序传递参数的
- ❑ 参数自右向左传递的目的也是保证生成汇编语言时,这些参数相对于**EBP**指向的栈位置的偏移量是固定的。



6.函数调用约定

□ **stdcall**调用者未对传入参数进行清理

○ **stdcall**是windows API默认方式，参数从右向左入栈

○ 通过对比在main函数**foo_stdcall**和**foo_cdecl**两种的函数调用代码，发现调用**foo_stdcall**相比**foo_cdecl**，缺少了**add esp, 0ch**指令，也就是说调用者未对传入参数进行清理



6.函数调用约定

```
.text:00401150      push     ebp
.text:00401151      mov      ebp, esp
.text:00401153      sub      esp, 40h
.text:00401156      push     ebx
.text:00401157      push     esi
.text:00401158      push     edi
.text:00401159      lea      edi, [ebp+var_40]
.text:0040115C      mov      ecx, 10h
.text:00401161      mov      eax, 0CCCCCCCCh
.text:00401166      rep stosl
.text:00401168      push     3
.text:0040116A      push     2
.text:0040116C      push     1
.text:0040116E      call     j__foo_cdecl
.text:00401173      add      esp, 0Ch
.text:00401176      push     3
.text:00401178      push     2
.text:0040117A      push     1
.text:0040117C      call     j__foo_stdcall@12 ; foo_stdcall(x,x,x)
.text:00401181      push     4
.text:00401183      push     3
.text:00401185      mov      edx, 2
.text:0040118A      mov      ecx, 1
.text:0040118F      call     j__foo_fastcall@12 ; foo_fastcall(x,x,x)
.text:00401194      xor      eax, eax
.text:00401196      pop      edi
.text:00401197      pop      esi
.text:00401198      pop      ebx
```

○查看main函数汇编代码



6.函数调用约定

@foo_stdcall反汇编代码

```
.text:00401090
.text:00401090      push     ebp
.text:00401091      mov      ebp, esp
.text:00401093      sub      esp, 40h
.text:00401096      push     ebx
.text:00401097      push     esi
.text:00401098      push     edi
.text:00401099      lea      edi, [ebp+var_40]
.text:0040109C      mov      ecx, 10h
.text:004010A1      mov      eax, 0CCCCCCCCh
.text:004010A6      rep stosd
.text:004010A8      mov      eax, [ebp+arg_8]
.text:004010AB      push     eax
.text:004010AC      mov      ecx, [ebp+arg_4]
.text:004010AF      push     ecx
.text:004010B0      mov      edx, [ebp+arg_0]
.text:004010B3      push     edx
.text:004010B4      push     offset Format ; "argv %d %d %d \n"
.text:004010B9      call     _printf
.text:004010BE      add      esp, 10h
.text:004010C1      pop      edi
.text:004010C2      pop      esi
.text:004010C3      pop      ebx
.text:004010C4      add      esp, 40h
.text:004010C7      cmp      ebp, esp
.text:004010C9      call     __chkesp
.text:004010CE      mov      esp, ebp
.text:004010D0      pop      ebp
.text:004010D1      retn     0Ch
.text:004010D1      _foo_stdcall@12 endp
```



6.函数调用约定

□ **foo_stdcall**采用的是**retn 0ch**的指令，而非**retn**。

- 该指令的作用是**retn + pop 0x0c**字节，即返回后使**ESP**增加**12**个字节，这与**foo_cdecl**的指令执行是一致的。
- **stdcall**方式的优点在于，被调用者函数内部存在清理参数代码，与调用函数后再执行**add esp, xxx**相比，代码尺寸要小，是**Win 32 API**库使用的函数调用约定。



6.函数调用约定

□fastcall方式

- 与stdcall方式基本类似，参数的清理也由被调用函数来负责
- 但该方式通常会使用寄存器（而非栈内存）去传递参数。
 - ❖若函数需要传递多个参数，会优先使用**ECX**和**EDX**来传递后两个参数，其余的参数再从右向左由栈传入。



```

.text:00401150
.text:00401150      push    ebp
.text:00401151      mov     ebp, esp
.text:00401153      sub     esp, 40h
.text:00401156      push    ebx
.text:00401157      push    esi
.text:00401158      push    edi
.text:00401159      lea     edi, [ebp+var_40]
.text:0040115C      mov     ecx, 10h
.text:00401161      mov     eax, 0CCCCCCCCh
.text:00401166      rep stosd
.text:00401168      push    3
.text:0040116A      push    2
.text:0040116C      push    1
.text:0040116E      call    sub_401005
.text:00401173      add     esp, 0Ch
.text:00401176      push    3
.text:00401178      push    2
.text:0040117A      push    1
.text:0040117C      call    sub_40100F
.text:00401181      push    4
.text:00401183      push    3
.text:00401185      mov     edx, 2
.text:0040118A      mov     ecx, 1
.text:0040118F      call    sub_401019
.text:00401194      xor     eax, eax
.text:00401196      pop     edi
.text:00401197      pop     esi
.text:00401198      pop     ebx
.text:00401199      add     esp, 40h
.text:0040119C      cmp     ebp, esp
.text:0040119E      call    sub_401240
.text:004011A3      mov     esp, ebp
.text:004011A5      pop     ebp
.text:004011A6      retn
.text:004011A6      endp

```

sub_401150



6.函数调用约定

□fastcall方式是很快的

- 由于**CPU**对寄存器的访问要快于对栈所在内存的访问速度，因此从函数调用本身来看，**fastcall**方式是很快的
- 但有时需要额外的系统开销来管理寄存器，如在调用函数前**ECX**、**EDX**中存有重要数据，那么需要先进行备份
- 此外，如果函数本身需要使用这两个寄存器，也需要将参数迁移到其他位置进行使用。





Q & A

谢谢!