程序的二进制表示

赵磊 副教授
武汉大学国家网络安全学院

主要内容

- 源代码与可执行文件
- 源代码的高级语义到指令的转换
  - X86体系结构及其指令集
  - 赋值、跳转、计算等源码语句转换
  - 函数/过程调用的实现

- 参考书目：
  - 《深入理解计算机系统（第三版）》

1. 程序的生成

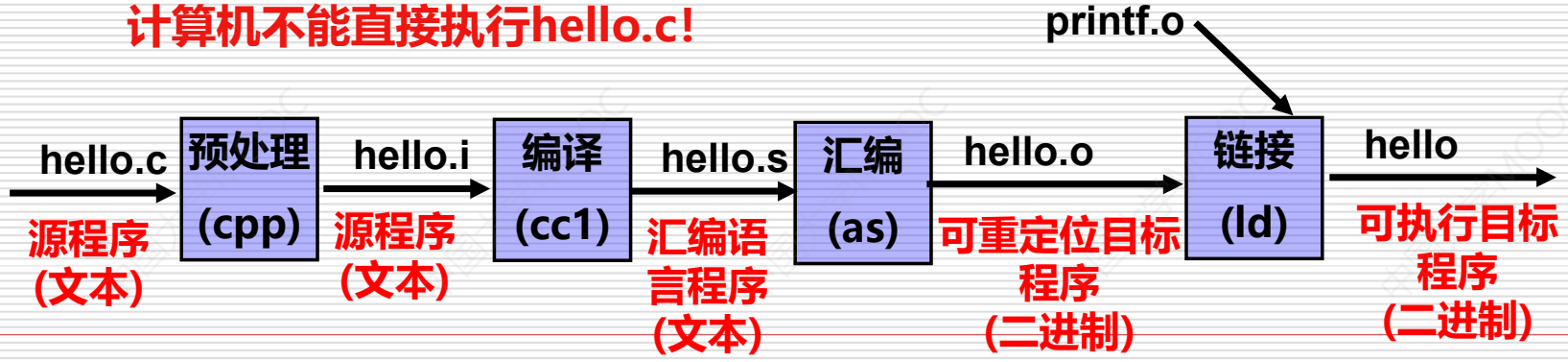
```
void main()
 {
    printf (“ hello world!\n”);
}
```

- 源代码程序是如何转换为可执行代码的？

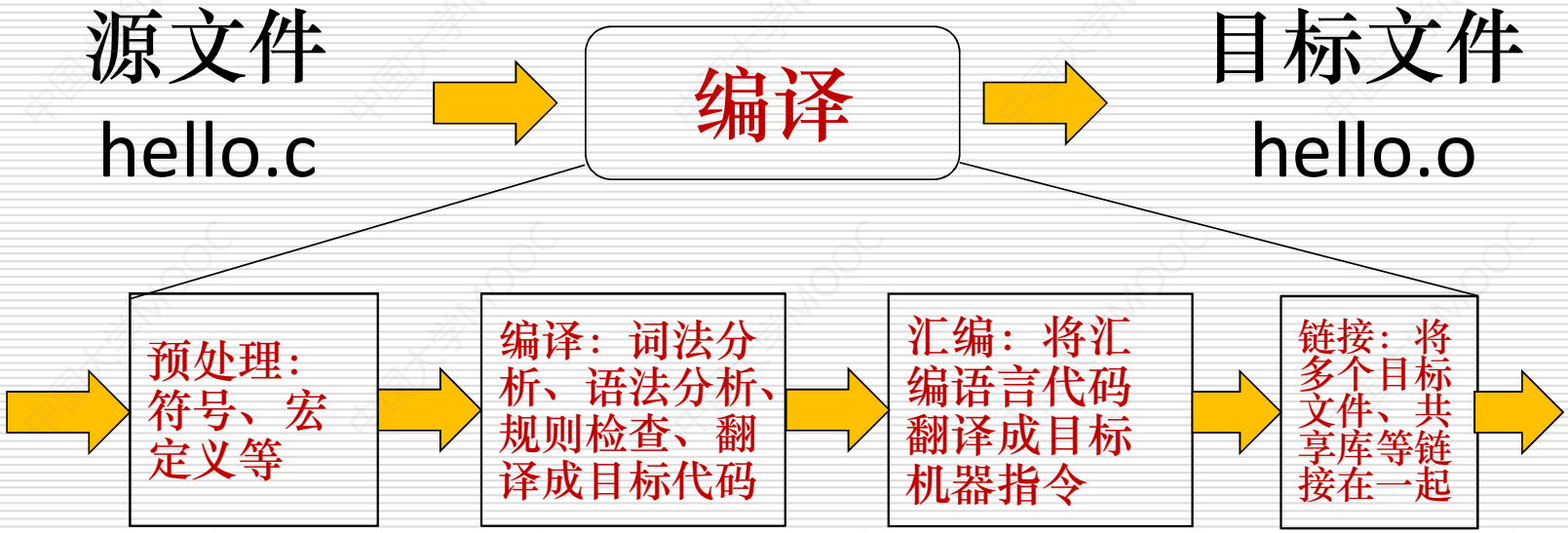
程序的转换处理过程

```
void main()
 {
    printf (“ hello world!\n”);
}
```

计算机不能直接执行hello.c!



程序的编译




```
void main()
{
    printf (" hello world!\n");
}
```

```
010110010101
101010101010
001101010110
010111101010
```

程序的编译

Dump of assembler code for function main:

0x08048394 <main+0>: lea 0x4(%esp),%ecx

0x08048398 <main+4>: and $0xfffffff0,%esp

0x0804839b <main+7>: pushl 0xfffffffc(%ecx)

0x0804839e <main+10>: push %ebp

0x0804839f <main+11>: mov %esp,%ebp

0x080483a1 <main+13>: push %ecx

目标文件中，无类型、无符号、无数据结构

gcc默认情况下中间文件不保留，直接生成目标文件
gcc –s 参数可以保留编译过程中的中间文件，比如as文件

源代码到二进制的转化

- 高级语言中的变量等在可执行文件中如何体现

```
01011001010
10101010101
00110101011
01011110101
```

指令
库函数的调用
(如printf)

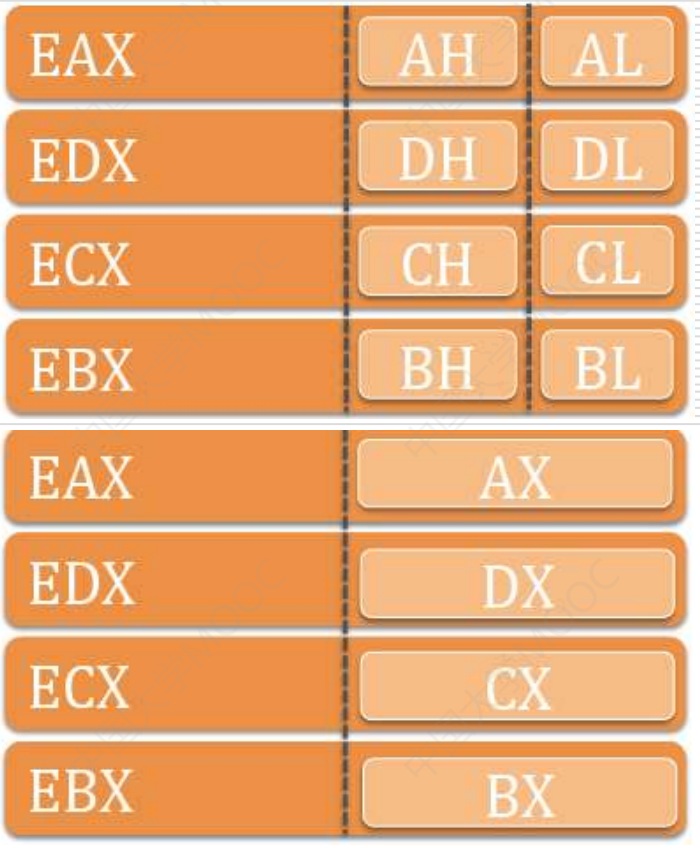
静态变量
全部变量等
如Hello world

2. 程序的二进制表示

- X86架构及其指令集
  - 寄存器
  - 通用寄存器
    - EAX/EDX/ECX/EBX/ESP/EBP/ESI/EDI
  - EIP寄存器
  - EFLAGS
    - 状态位

通用寄存器的访问

- 低8位
- 中8位
- 低16位
- 全32位



寄存器操作

- AT&T语法
  - Linux & gcc 默认的汇编语法
  - 与intel的区别：源与目的操作数方向相反

指令	语义
mov %ebx, %eax	将ebx的值移动到eax
add %ebx, %eax	%eax = %edx + %eax
shl $2, %ecx	%ecx = %ecx << 2

IA-32常用指令类型

- 传送指令
  - 通用数据传送指令
    - MOV：一般传送，包括movb、movw和movl等
    - XCHG：数据交换
  - 地址传送指令
    - LEA：加载有效地址，如leal (%edx,%eax),

“入栈”和“出栈”操作

- 采用“先进后出”方式进行访问的一块存储区，用于嵌套过程调用
  - Push
  - Pop
  - Call
  - Ret
  - leave

定点算术运算指令汇总

指令	显式操作数	影响的常用标志	操作数类型	AT&T 指令助记符	对应 C 运算符
ADD	2 个	OF、ZF、SF、CF	无/带符号整数	addb、addw、addl	+
SUB	2 个	OF、ZF、SF、CF	无/带符号整数	subb、subw、subl	−
INC	1 个	OF、ZF、SF	无/带符号整数	incb、incw、incl	++
DEC	1 个	OF、ZF、SF	无/带符号整数	decb、decw、decl	−−
NEG	1 个	OF、ZF、SF、CF	无/带符号整数	negb、negw、negl	−
CMP	2 个	OF、ZF、SF、CF	无/带符号整数	cmpb、cmpw、cmpl	<, <=, >, >=
MUL	1 个	OF、CF	无符号整数	mulb、mulw、mull	*
MUL	2 个	OF、CF	无符号整数	mulb、mulw、mull	*
MUL	3 个	OF、CF	无符号整数	mulb、mulw、mull	*
IMUL	1 个	OF、CF	带符号整数	imulb、imulw、imull	*
IMUL	2 个	OF、CF	带符号整数	imulb、imulw、imull	*
IMUL	3 个	OF、CF	带符号整数	imulb、imulw、imull	*
DIV	1 个	无	无符号整数	divb、divw、divl	/, %
IDIV	1 个	无	带符号整数	idivb、idivw、idivl	/, %

程序的机器级表示

- 高级语言语义转换为低级别的指令操作
  - 指令：操作码
  - 操作数：内存单元、寄存器
  - 函数调用
    - 栈帧的切换
    - 返回地址、栈平衡的维护
    - 局部变量的分配、传递

x86下的内存寻址

- x86支持字节可寻址
  - 基本的存储单元就是一个字节
  - mov [al], dl
  - 内存可视作线性数组，而内存地址就是数组的下标
- 其它：字可寻址
  - 对于32位大小的字，一次寻址可以得到4字节
  - 内存地址必须以4对齐
    - 错误的寻址地址：address 0x6

内存地址

- 指令的操作数只有寄存器或内存地址
- 编译器负责把高级语言层面的变量操作转化为地址操作
  - int buf[16];
  - buf[2]
  - *(buf + 2)

  - addr = (unsigned int) buf;
  - *(addr + sizeof(int) * 2)

- addr = (unsigned int) buf;
  - lea -0x16(%ebp), %eax
- *(addr + sizeof(int) * 2)
  - add $8, %eax
  - mov (%eax), %edx
- *buf
  - mov -0x16(%ebp), %eax

控制流

- 高级语义
  - if – else
  - while
  - for
  - switch
- 低级语义
  - jump
  - branch

控制流

- 跳转
  - 直接跳转：jmp 0x45
  - 间接跳转：jmp eax
- 分支
  - if (eflags) jmp x

- if (x <=y)
- z = x;
- else
  - z = y ;

- 计算x－y，并设置EFLAGS
  - CF if x < y
  - ZF if x==y
- 测试EFLAGS, 如果CF和ZF都没有被置位，跳转到z=y
- 否则跳转到z = x

条件跳转

- JE; 等于则跳转
- JNE 不等于则跳转
- JZ ;为 0 则跳转
- JNZ ;不为 0 则跳转
- JS ;为负则跳转
- JNS ;不为负则跳转
- JC ;进位则跳转
- JNC ;不进位则跳转
- JO ;溢出则跳转
- JNO ;不溢出则跳转

结构、类、对象等

- 复杂结构的表示方法
  - 结构体
  - 枚举类型等
- C++类及对象的表示
  - 对象的内存布局
  - 虚函数/虚函数表

3. 函数的实现

```
int gray ( int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
         c = a;
   else
         c = b;
   d = red(c, buf);
   return d;
}
```

如何访问传递来的参数

局部变量

如何向被调用者传参

返回值如何返回

函数在内存中的实现

- 编译器在栈上实现函数（过程）
  - 遵循call / ret的调用规则
- 编译器通过给函数分配栈帧（stack frame)来保存上述变量或参数
  - 每个函数有其独立的栈（不考虑编译优化的规则）
  - 后进先出：如果A调用B，则函数B的栈帧先于A的栈帧弹出

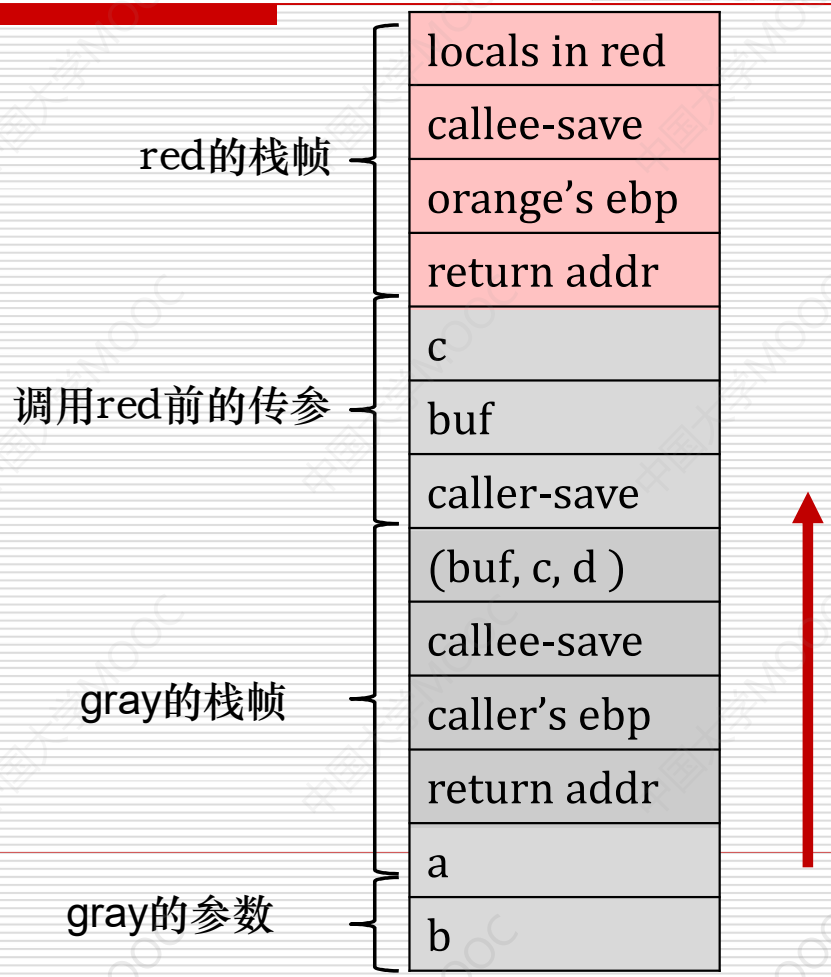
函数调用方式

cdecl	stdcall
C/C++缺省的调用方式 Linux/gcc缺省调用方法	Pascal程序的缺省调用方式 Win32 API的缺省调用方式
参数从右到左压栈	参数从右到左压栈
调用者函数维护栈的平衡	被调用者函数维护栈的平衡

调用者寄存器（Caller Save）: eax, edx, ecx
被调用者寄存器（Callee Save）: ebx, esi, edi, ebp, esp
堆栈的平衡：esp 和 ebp
返回值：通常放在eax

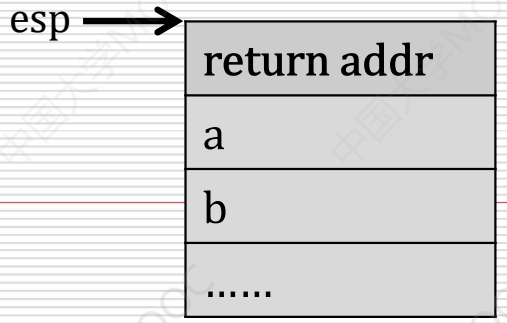
cdecl

```
int gray( int a, int b)
{
   char buf[16];
   int c, d;
   if(a > b)
         c = a;
   else
         c = b;
   d = red(c, buf);
   return c;
}
```



当调用gray函数时

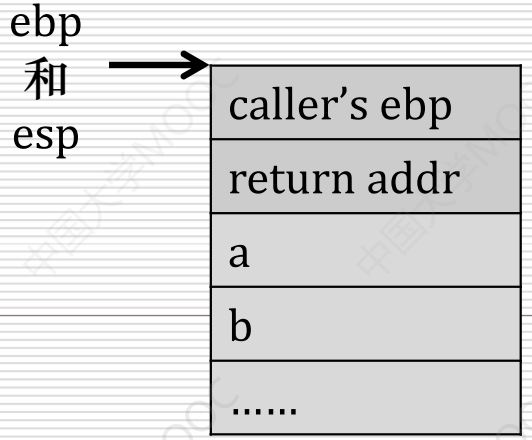
1. 返回地址入栈
   （调用gray的下一条指令地址）



当调用gray函数时

1. 返回地址入栈
   （调用gray的下一条指令地址）

2. 构建自己的栈帧
   -- 把调用者ebp压栈
   -- 把当前esp赋值给ebp
   -- 此时第一个参数的地址为ebp+8

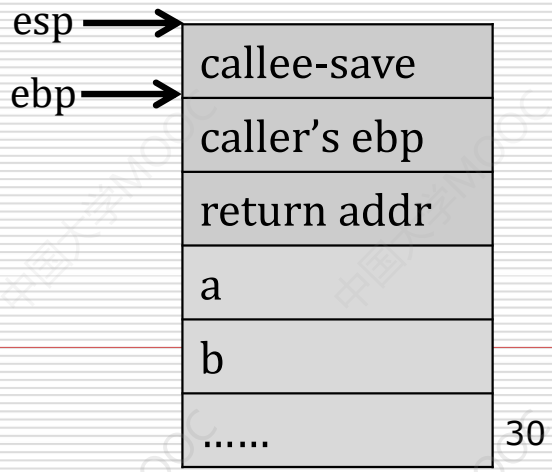


当调用gray函数时

1. 返回地址入栈
   （调用gray的下一条指令地址）

2. 构建自己的栈帧
   -- 把调用者ebp压栈
   -- 把当前esp赋值给ebp
   -- 此时第一个参数的地址为ebp+8

3. 保存被调用者寄存器
   -- edi, esi, ebx



30

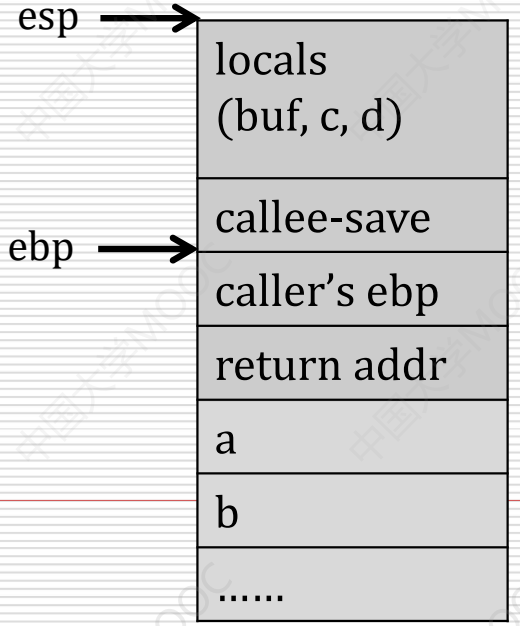
当调用gray函数时

1. 返回地址入栈
   （调用gray的下一条指令地址）

2. 构建自己的栈帧
   -- 把调用者ebp压栈
   -- 把当前esp赋值给ebp
   -- 此时第一个参数的地址为ebp+8

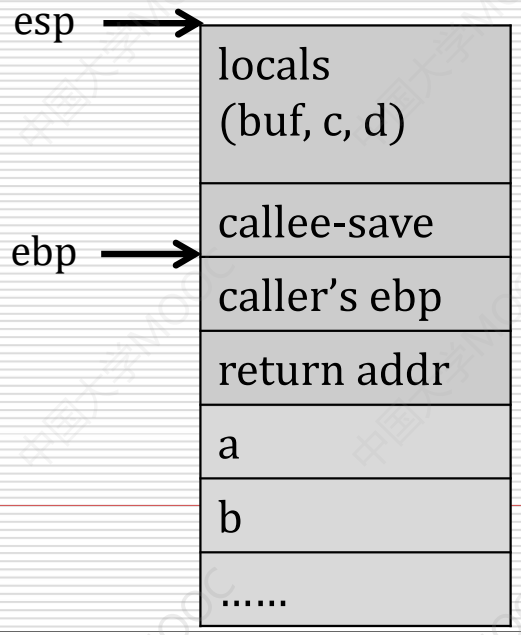
3. 保存被调用者寄存器
   -- edi, esi, ebx

4. 为局部变量分配空间
   -- esp减一个数即为分配的局部变量空间



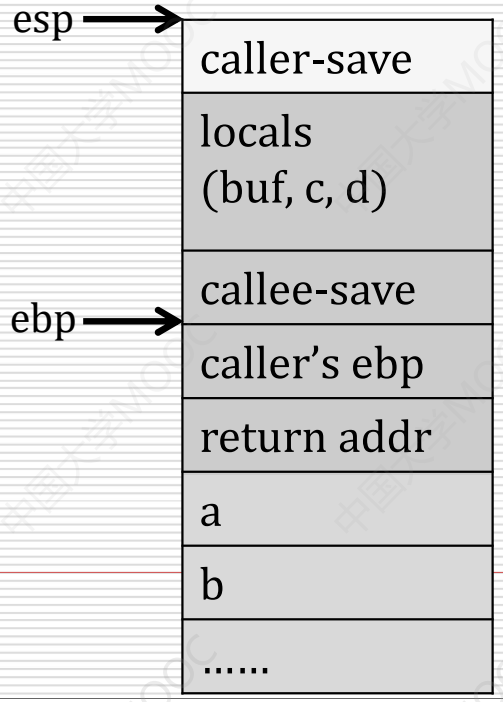
31

对于调用者gray调用red



对于调用者gray调用red

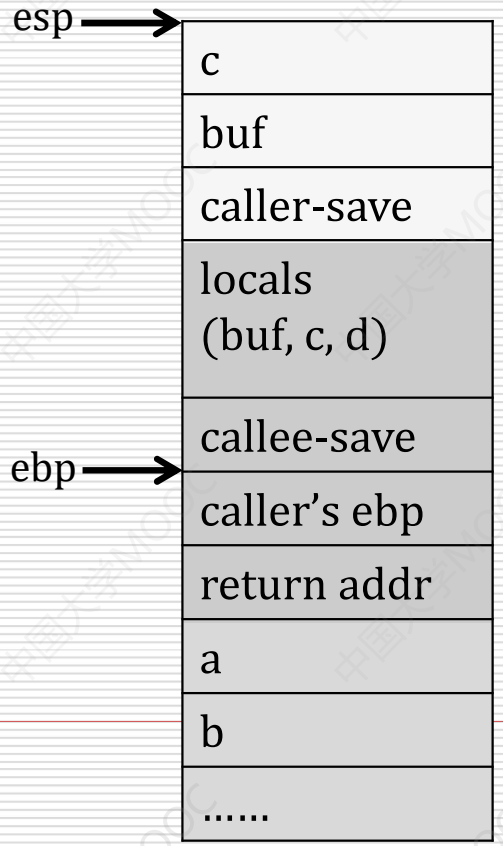
1. 把调用者寄存器保存到栈中
   -- eax, edx, ecx
   -- red返回后， gray可能用到



对于调用者gray调用red

1. 把调用者寄存器保存到栈中
   -- eax, edx, ecx
   -- red返回后， gray可能用到

2. 把red的参数由右向左依次入栈

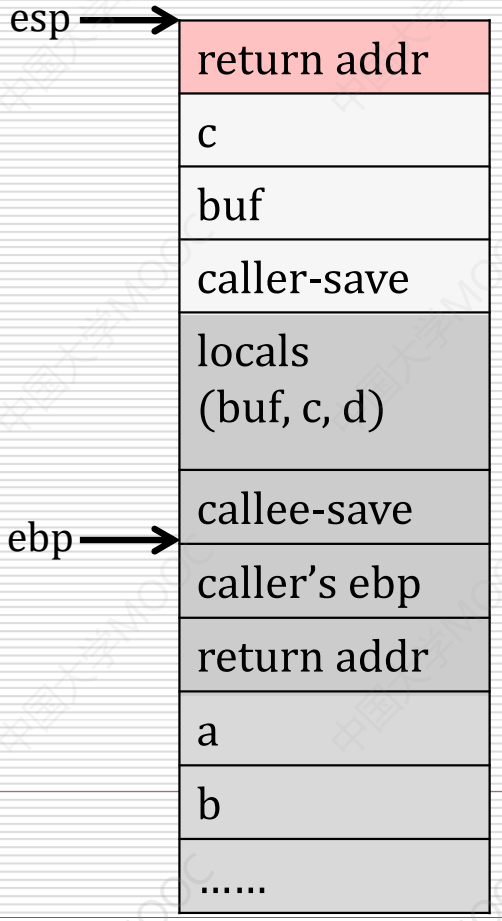


对于调用者gray调用red

1. 把调用者寄存器保存到栈中
   -- eax, edx, ecx
   -- red返回后， gray可能用到

2. 把red的参数由右向左依次入栈

3. 返回地址入栈，即red执行结束后
   下一条指令的地址



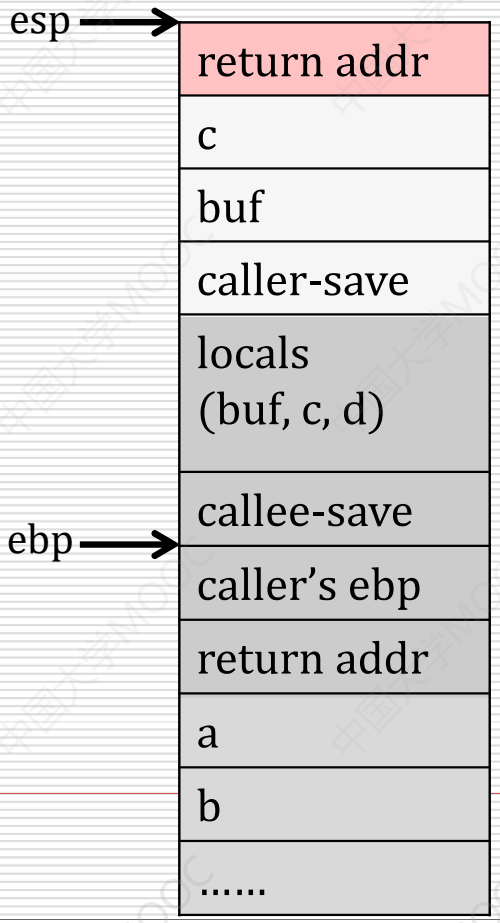
对于调用者gray调用red

1. 把调用者寄存器保存到栈中
   -- eax, edx, ecx
   -- red返回后， gray可能用到

2. 把red的参数由右向左依次入栈

3. 返回地址入栈，即red执行结束后
   下一条指令的地址

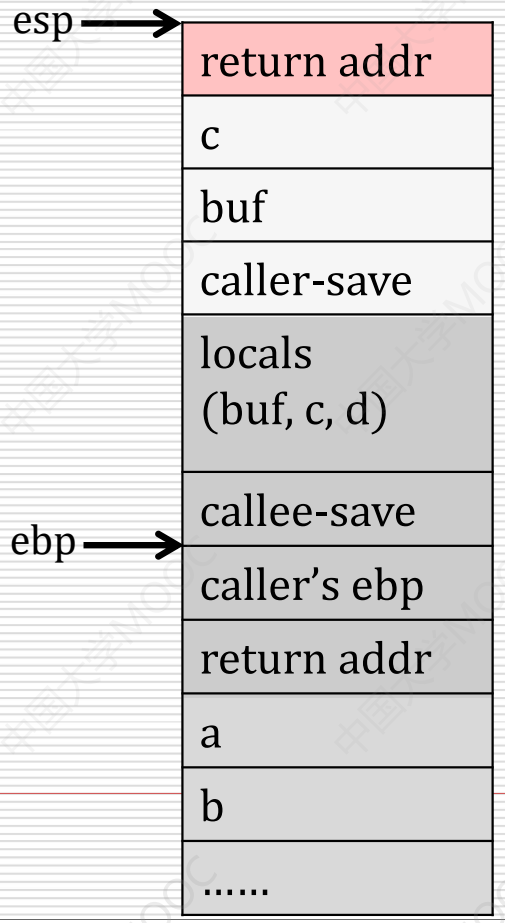
4. 跳转到red



36

当red返回时

1. 如果有返回值，则保存它，一般在eax中
2. 回收局部变量空间
3. 恢复被调用者寄存器的值
4. 恢复gray的栈帧
   -- pop ebp

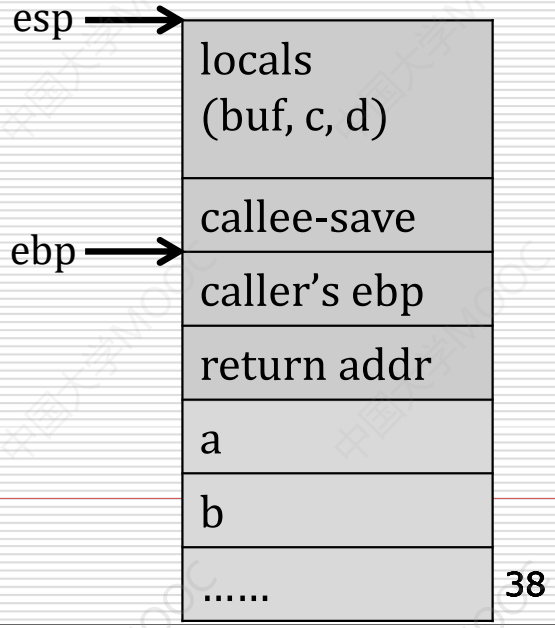


37

当red返回时

5. 返回并把控制权交于gray
   -- ret
   -- 弹出返回地址并跳至返回地址处执行

6. 维护栈平衡
   --清除red的参数
   --恢复所有caller-save寄存器
   …



38

思考题

- 根据程序的二进制表示，思考缓冲区溢出的原理