弱类型安全的编程语言

- 高级编程语言支持多类型（int, char, string等）
- 程序执行时（二进制），只看到寄存器和内存指针
- 弱类型编程语言：C/C++/汇编
  - 支持隐类型转换，仅在编译时检测部分类型冲突
  - 指针操作很任性
  - 大大提高运行效率，但是引入类型冲突的隐患

- 类型安全的编程语言：JAVA/C#
  - 不支持隐类型转换
  - 引用代替指针（支持大小、长度等运行时检查）

弱类型安全的编程语言

武漢大學 WUHAN UNIVERSITY

类型冲突：内存破坏漏洞的根本

漏洞利用的充分条件：任意（OS字长）地址写

```
01 #include<string.h>
02
03 void *
04 memcpy(void *s1, const void *s2, register size_t n)
05 {
06     register char *p1 = s1;
07     register const char *p2 = s2;
08
09     if (n) {
10         n++;
11         while (--n > 0) {
12             *p1++ = *p2++;
13         }
14     }
15     return s1;
16 }
```

高级语言中的类型转换

- 变量
  - 全局变量
    - 数据段
  - 局部变量
    - 栈区
    - 由编译器依据其类型确定大小
    - 随程序的执行过程动态创建
  - 动态分配的变量
    - 堆区
    - 用户管理 malloc/free, new delete

整形

- 补码
  - 补码的优势
- 无符号数与有符号数的转换
  - C支持互转
  - 对CPU计算单元却是统一的
  - 整形溢出问题

整形溢出示例


```
#include <stdio.h>

void main(){
    int sa = 0x7FFFFFFF;
    unsigned int ua = 0x7FFFFFFF;

    sa = sa + 2;
    ua = ua + 2;
    printf("size of sa is %d\n", sizeof(int));
    printf("sa is %d\n", sa);
    printf("ua is %d\n", ua);
    if (sa < 1)
        printf("sa is overflow\n");
    if (ua < 1)
        printf("ua is overflow\n");
}
```

整形溢出示例


```
#include <stdio.h>

void main(){
    int sa = 0x7FFFFFFF;
    unsigned int ua = 0x7FFFFFFF;

    sa = sa + 2;
    ua = ua + 2;
    printf("size of sa is %d\n", sizeof(int));
    printf("sa is %d\n", sa);
    printf("ua is %d\n", ua);
    if (sa < 1)
        printf("sa is overflow\n");
    if (ua < 1)
        printf("ua is overflow\n");
}
```

```
leizhao@ubuntu:~/Desktop$ ./test_unsigned
size of sa is 4
sa is -2147483647
ua is -2147483647
sa is overflow
leizhao@ubuntu:~/Desktop$
```

整形溢出示例

```
#include <stdio.h>

void main(){
    int sa = 0x7FFFFFFF;
    unsigned int ua = 0x7FFFFFFF;

    sa = sa + 2;
    ua = ua + 2;
    printf("size of sa is %d\n", sizeof(i
    printf("sa is %d\n", sa);
    printf("ua is %d\n", ua);
    if (sa < 1)
        printf("sa is overflow\n");
    if (ua < 1)
        printf("ua is overflow\n");
}
```

```
0804844d <main>:
 804844d:   55                      push   %ebp
 804844e:   89 e5                   mov    %esp,%ebp
 8048450:   83 e4 f0                and    $0xfffffff0,%esp
 8048453:   83 ec 20                sub    $0x20,%esp
 8048456:   c7 44 24 18 ff ff ff 7f movl   $0x7fffffff,0x18(%esp)
 804845e:   c7 44 24 1c ff ff ff 7f movl   $0x7fffffff,0x1c(%esp)
 8048466:   83 44 24 18 02          addl   $0x2,0x18(%esp)
 804846b:   83 44 24 1c 02          addl   $0x2,0x1c(%esp)
 8048470:   c7 44 24 04 04 00 00    movl   $0x4,0x4(%esp)
 8048478:   c7 04 24 70 85 04 08    movl   $0x8048570,(%esp)
 804847f:   e8 8c fe ff ff          call   8048310 <printf@plt>
 8048484:   8b 44 24 18             mov    0x18(%esp),%eax
 8048488:   89 44 24 04             mov    %eax,0x4(%esp)
 804848c:   c7 04 24 82 85 04 08    movl   $0x8048582,(%esp)
 8048493:   e8 78 fe ff ff          call   8048310 <printf@plt>
 8048498:   8b 44 24 1c             mov    0x1c(%esp),%eax
 804849c:   89 44 24 04             mov    %eax,0x4(%esp)
 80484a0:   c7 04 24 8c 85 04 08    movl   $0x804858c,(%esp)
 80484a7:   e8 64 fe ff ff          call   8048310 <printf@plt>
 80484ac:   83 7c 24 18 00          cmpl   $0x0,0x18(%esp)
 80484b1:   7f 0c                   jg     80484bf <main+0x72>
 80484b3:   c7 04 24 96 85 04 08    movl   $0x8048596,(%esp)
 80484ba:   e8 61 fe ff ff          call   8048320 <puts@plt>
 80484bf:   83 7c 24 1c 00          cmpl   $0x0,0x1c(%esp)
 80484c4:   75 0c                   jne    80484d2 <main+0x85>
 80484c6:   c7 04 24 a5 85 04 08    movl   $0x80485a5,(%esp)
 80484cd:   e8 4e fe ff ff          call   8048320 <puts@plt>
 80484d2:   c9                      leave
 80484d3:   c3                      ret
```

C语言程序中的整数

无符号数 unsigned int (short / long)；带符号整数：int (short / long)

常在一个数的后面加一个“u”或“U”表示无符号数

若同时有无符号和带符号整数，则C编译器将带符号整数强制转换为无符号数

假定以下关系表达式在32位用补码表示的机器上执行，结果是什么？

关系表达式	运算类型	结果	说明
0 == 0U			
-1 < 0			
-1 < 0U			
2147483647 > -2147483647-1			
2147483647U > -2147483647-1			
2147483647 > (int) 2147483648U			
-1 > -2			
(unsigned) -1 > -2			

C语言程序中的整数

关系 表达式	类型	结果	说明
0 = = 0U	无	1	00…0B = 00…0B
-1 < 0	带	1	11…1B (-1) < 00…0B (0)
-1 < 0U	无	0*	11…1B (2^{32}-1) > 00…0B(0)
2147483647 > -2147483647 - 1	带	1	011…1B (2^{31}-1) > 100…0B (-2^{31})
2147483647U > -2147483647 - 1	无	0*	011…1B (2^{31}-1) < 100…0B(2^{31})
2147483647 > (int) 2147483648U	带	1*	011…1B (2^{31}-1) > 100…0B (-2^{31})
-1 > -2	带	1	11…1B (-1) > 11…10B (-2)
(unsigned) -1 > -2	无	1	11…1B (2^{32}-1) > 11…10B (2^{32}-2)

带*的结果与常规预想的相反！

C语言程序中的整数

例如，考虑以下C代码：

```
int x = –1;
unsigned u = 2147483648;
printf ( “x = %u = %d\n” , x, x);
printf ( “u = %u = %d\n” , u, u);
```

在32位机器上运行上述代码时，它的输出结果是什么？为什么？

x = 4294967295 = –1

u = 2147483648 = –2147483648

- –1的补码整数表示为“11…1”，作为32位无符号数解释时，其值为 $2^{32}–1$= 4 294 967 296–1 = 4 294 967 295。
- 2^{31}的无符号数表示为“100…0”，被解释为32位带符号整数时，其值为最小负数：$–2^{32-1} = –2^{31}$ = –2 147 483 648。

C语言程序中的整数

1）在有些32位系统上，C表达式-2147483648 < 2147483647的执行结果为false。Why?

2）若定义变量“int i=-2147483648;”，则“i < 2147483647”的执行结果为true。Why?

3）如果将表达式写成“-2147483647-1 < 2147483647”，则结果会怎样呢? Why?

1）在ISO C90标准下，2147483648为unsigned int型，因此“-2147483648 < 2147483647”按无符号数比较，10……0B比01……1大，结果为false。

在ISO C99标准下，2147483648为long long型，因此“-2147483648 < 2147483647”按带符号整数比较，10……0B比01……1小，结果为true。

2）i < 2147483647 按int型数比较，结果为true。

3）-2147483647-1 < 2147483647 按int型比较，结果为true。

编译器处理常量时默认的类型

- C90

范围	类型
$0 \sim 2^{31}-1$	int
$2^{31} \sim 2^{32}-1$	unsigned int
$2^{32} \sim 2^{63}-1$	long long
$2^{63} \sim 2^{64}-1$	unsigned long long

2^{31}=2147483648，100…0（31个0）

- C99

范围	类型
$0 \sim 2^{31}-1$	int
$2^{31} \sim 2^{63}-1$	long long
$2^{63} \sim 2^{64}-1$	unsigned long long

整形溢出

- 无符号数加法溢出
- 有符号数加法溢出
- 有符号转换为无符号时溢出

整形溢出的检查

- 如何判断两个无符号数相加的溢出

$$\text{result}=\begin{cases} x+y & (x+y<2^n) \\ x+y-2^n & (2^n \leqslant x+y<2^{n+1}) \end{cases}$$

发生溢出时，一定满足 result<x and result<y

否则，若$x+y-2^n \geq x$，则 $y \geq 2^n$，这是不可能的!

整形溢出的检查

- 如何判断两个无符号数相减的溢出

整形溢出的检查

- 如何判断两个有符号数相加的溢出

$$\text{result}=\begin{cases} x+y-2^{n} & (2^{n-1}\leqslant x+y) \quad \text{正溢出} \\ x+y & (-2^{n-1}\leqslant x+y<2^{n-1}) \quad \text{正常} \\ x+y+2^{n} & (x+y<-2^{n-1}) \quad \text{负溢出} \end{cases}$$

CF=0, ZF=0, OF=1, SF=1

CF=1, ZF=0, OF=1, SF=0

```
int tadd_ok(int x, int y) {
    int sum = x+y;
    int neg_over = x < 0 && y < 0 && sum >= 0;
    int pos_over = x >= 0 && y >= 0 && sum < 0;
    return !neg_over && !pos_over;
}
```

整形溢出的检查

- 如何判断两个有符号数相减的溢出

$$\text{result}=\begin{cases} x-y-2^{n} & (2^{n-1}\leqslant x-y) & \text{正溢出} \\ x-y & (-2^{n-1}\leqslant x-y<2^{n-1}) & \text{正常} \\ x-y+2^{n} & (x-y<-2^{n-1}) & \text{负溢出} \end{cases}$$

带符号整数减

```
int tsub_ok(int x, int y) {
        return tadd_ok(x, -y);
}
```

当x>=0，y=0x80000000时，该函数判断错误

整形溢出的检查

- 如何判断两个有符号数相减的溢出
  - 何种情况下溢出?
  - 0000 – 1000 = 1000
  - 1000 – 0001 = 0111

整形溢出的检查

- 如何判断两个有符号数相减的溢出
  - x y的符号不同，且 x 与 r 的符号不同时溢出

```
int subOK(int x, int y) {
   int diff = x+~y+1;
   int x_neg = x>>31;
   int y_neg = y>>31;
   int d_neg = diff>>31;
   return !(~(x_neg ^ ~y_neg) & (x_neg ^ d_neg));
}
```

浮点数编码

- 浮点数的表示方法：IEEE 754标准
- 浮点数表示的精度问题

浮点数精度问题示例

- 1991年海湾战争，美国爱国者反导系统在拦击伊拉克导弹时击中美军的一个兵营，造成28名士兵死亡，底层的原因是一个浮点数计算的不精确。

浮点数精度问题示例

- 爱国者反导系统内以计数器来实现时钟，每0.1秒增加1
- 系统以0.1为精度，乘以计数值来生成以秒为单位的时间
  - 程序将0.1表示为一个24位的二进制小数
  - 0.1的二进制位一个无穷序列，$0.0[0011]\ [0011]_{2\ldots\ldots}$
  - 程序内的表示只取小数点后的前23位，记为x
  - 0.1-x的二进制表示是多少
  - 0.1-x的近似十进制是多少
  - 系统启动时，计算器从0开始，若系统已经运行了100个小时，程序计算出的时间与实际实现相差多少
  - 系统根据导弹速度和被检测到的时间来预测飞到哪里，则预测偏差是多少

浮点数精度问题示例

- 0.1-x的二进制表示是多少
  - $0.1 - x = 0.000[1100][1100]\ [1100][1100]\ [1100] - 0.000[1100][1100]_{2.....}$
  - $0.1 - x = 2^{-23} * \ 0.[1100]_{2.....} = 2^{-20} * \ 0.000[1100]_{2.....}$
- 0.1-x的近似十进制是多少
  - $0.1 - x = 2\text{-}20 * \ 0.1$
  - 约等于 $9.54 * 10^{-8}$
- 系统启动时，计算器从0开始，若系统已经运行了100个小时，程序计算出的时间与实际实现相差多少
  - $9.54 * 10\text{-}8 \ * 100 * 3600 * 10 = 0.343$秒
- 若导弹速度为2000米每秒，则预测偏差是多少
  - 687米

3) 字符

- 字符的编码：
  - ASCII： 0~255,1B
  - Unicode：用于各种语言文字，2B（高位补0）
- 汉字编码
  - GB2312-80:94区*94位，共6763个汉字
  - 01-09区为特殊符号
  - 16-55区为一级汉字，按拼音排序
  - 56-87区为二级汉字，按部首/笔画排序
  - 区码和位码分别+0xA0构成存储表示

3) 字符

- 字符串
  - 长度值+字符串: String
  - 字符串+结束符（’\0’）: char*
- ASCII与Unicode的转换
  - typedef unsigned short wchar_t，宽字符
  - MultiByteToWideChar()和WideCharToMultiByte()
  - 在参数传递、赋值时注意类型

4) 布尔类型

- bool
  - 1B
  - 0：非0
  - 可以用整数或者字符代替
- 逆向还原时比较简单

5) 地址与指针

- 地址(以32位为例)
  - 0x0000000~0xffffffff
  - 每一个地址存放1个字节的数据或代码
  - 变量名仅仅是内存地址的一个别名，或者记号
  - &算子为获取变量的地址
- 指针-32位
  - TYPE *
  - 该数据类型的变量保存一个地址

5) 地址与指针

- 地址与指针的区别
  - int a=10; //地址不可修改（常量）
  - int* ipt=&a;//指针的值可以修改（变量）

- 指针的类型不同，对相同地址的数据的解释与操作不同（谁负责）

- 指针的操作：加和减
  - 不同类型的指针加减操作所得的地址偏移不同

6) 常量

- 常量
  - 固定不变
  - 位于特定的区域
  - 一样有地址存放

- #define 替换常量，字符串存在于数据区，整数为立即数

- const 修饰变量，可用使用，位于数据区

作用域、生命期、存储空间

武漢大學
WUHAN UNIVERSITY

- 目标文件中的数据节
  - .DATA节
    - 已初始化的全局变量
  - .BSS节
    - 未初始化的全局变量
  - .RODATA节
    - Read Only Data
    - 常量

作用域与存储空间

- 局部变量
  - 动态分配
- const修饰的变量/常量
- 全局变量
  - 静态变量
  - 静态局部变量
  - 无static的全局变量

示例1

- 以下符号的区别?

```
# include <stdio.h>

static int global_i = 2000;
int csa = 200;
```

示例1

- 以下符号的区别?

```
# include <stdio.h>

static int global_i = 2000;
int csa = 200;
```

- 作用域不同
  - 全局符号
    - 可供其它模块引用（extern）
  - 局部符号
    - 本模块内引用
  - 链接器检查

示例2

- 以下定义的区别?

```
# include <stdio.h>

static int          si = 2000;
const int           ci = 256;
static const int    sci = 200;
```

示例2

- 以下定义的区别?

```
# include <stdio.h>

static int          si = 2000;
const int           ci = 256;
static const int    sci = 200;
```

```
8048480:    a1 24 a0 04 08          mov    0x804a024,%eax
8048485:    89 44 24 04             mov    %eax,0x4(%esp)
8048489:    c7 04 24 83 87 04 08    movl   $0x8048783,(%esp)
8048490:    e8 7b fe ff ff          call   8048310 <printf@plt>
8048495:    b8 00 01 00 00          mov    $0x100,%eax
804849a:    89 44 24 04             mov    %eax,0x4(%esp)
804849e:    c7 04 24 9f 87 04 08    movl   $0x804879f,(%esp)
80484a5:    e8 66 fe ff ff          call   8048310 <printf@plt>
80484aa:    b8 c8 00 00 00          mov    $0xc8,%eax
80484af:    89 44 24 04             mov    %eax,0x4(%esp)
80484b3:    c7 04 24 9f 87 04 08    movl   $0x804879f,(%esp)
80484ba:    e8 51 fe ff ff          call   8048310 <printf@plt>
```

示例2

- 以下定义的区别?

```
# include <stdio.h>

static char          si[] = “2000”;
const char           ci[] = “256”;
static const char    sci[] = “200”;
```

示例2

- 以下定义的区别?

```
# include <stdio.h>

static char          si[] = “2000”;
const char           ci[] = “256”;
static const char    sci[] = “200”;
```

```
8048480:    c7 44 24 04 24 a0 04    movl   $0x804a024,0x4(%esp)
8048487:    08
8048488:    c7 04 24 73 87 04 08    movl   $0x8048773,(%esp)
804848f:    e8 7c fe ff ff          call   8048310 <printf@plt>
8048494:    c7 44 24 04 50 87 04    movl   $0x8048750,0x4(%esp)
804849b:    08
804849c:    c7 04 24 8f 87 04 08    movl   $0x804878f,(%esp)
80484a3:    e8 68 fe ff ff          call   8048310 <printf@plt>
80484a8:    c7 44 24 04 54 87 04    movl   $0x8048754,0x4(%esp)
80484af:    08
80484b0:    c7 04 24 8f 87 04 08    movl   $0x804878f,(%esp)
80484b7:    e8 54 fe ff ff          call   8048310 <printf@plt>
```

示例2

- 以下定义的区别?

```
# include <stdio.h>

static char          si[] = “2000”;
const char           ci[] = “256”;
static const char    sci[] = “200”;
```

```
8048480:    c7 44 24 04 24 a0 04    movl   $0x804a024,0x4(%esp)
8048487:    08
8048488:    c7 04 24 73 87 04 08    movl   $0x8048773,(%esp)
804848f:    e8 7c fe ff ff          call   8048310 <printf@plt>
8048494:    c7 44 24 04 50 87 04    movl   $0x8048750,0x4(%esp)
804849b:    08
804849c:    c7 04 24 8f 87 04 08    movl   $0x804878f,(%esp)
80484a3:    e8 68 fe ff ff          call   8048310 <printf@plt>
80484a8:    c7 44 24 04 54 87 04    movl   $0x8048754,0x4(%esp)
80484af:    08
80484b0:    c7 04 24 8f 87 04 08    movl   $0x804878f,(%esp)
80484b7:    e8 54 fe ff ff          call   8048310 <printf@plt>
```

```
[15] .rodata           PROGBITS        08048748 000748 0000fc
[16] .eh_frame_hdr     PROGBITS        08048844 000844 000034
```

```
[24] .data             PROGBITS        0804a01c 00101c 000014
[25] .bss              NOBITS          0804a030 001030 000004
```

示例2

- 以下定义的区别?

```
# include <stdio.h>

static int          si = 2000;
const int           ci = 256;
static const int    sci = 200;
```

- 全局变量
  - static: 模块内全局变量
    - DATA/BSS
  - const: 编译器转为常量
    - rodata

示例3

- 以下定义的区别?

```
# include <stdio.h>

const int ci = 200;

void main() {
    const int mci = 100;
```

示例3

- 以下定义的区别?

```
# include <stdio.h>

const int ci = 200;

void main() {
    const int mci = 100;
```

```
80484aa:    b8 c8 00 00 00          mov    $0xc8,%eax
80484af:    89 44 24 04             mov    %eax,0x4(%esp)
80484b3:    c7 04 24 9f 87 04 08    movl   $0x804879f,(%esp)
80484ba:    e8 51 fe ff ff          call   8048310 <printf@plt>
80484bf:    c7 44 24 18 64 00 00    movl   $0x64,0x18(%esp)
80484c6:    00
80484c7:    8b 44 24 18             mov    0x18(%esp),%eax
80484cb:    89 44 24 04             mov    %eax,0x4(%esp)
80484cf:    c7 04 24 ba 87 04 08    movl   $0x80487ba,(%esp)
80484d6:    e8 35 fe ff ff          call   8048310 <printf@plt>
```

示例3

- 以下定义的区别?
  - const 修饰的局部变量在空间分配上与普通局部变量一样
  - const 局部变量的值不可修改
    - 编译时检查

```
# include <stdio.h>

const int ci = 200;

void main() {
    const int mci = 100;
```

```
80484aa:    b8 c8 00 00 00          mov    $0xc8,%eax
80484af:    89 44 24 04             mov    %eax,0x4(%esp)
80484b3:    c7 04 24 9f 87 04 08    movl   $0x804879f,(%esp)
80484ba:    e8 51 fe ff ff          call   8048310 <printf@plt>
80484bf:    c7 44 24 18 64 00 00    movl   $0x64,0x18(%esp)
80484c6:    00
80484c7:    8b 44 24 18             mov    0x18(%esp),%eax
80484cb:    89 44 24 04             mov    %eax,0x4(%esp)
80484cf:    c7 04 24 ba 87 04 08    movl   $0x80487ba,(%esp)
80484d6:    e8 35 fe ff ff          call   8048310 <printf@plt>
```

静态变量示例

- 全局静态变量与局部静态变量

```
void func()
{
  static int local_i = 1000;
  printf("local static variable: %d\n", local_i);
  local_i++;
}

Void main() {
....
 for (sa = 256; sa > 252; sa--) {
    printf("for loop\n");    func();
}}
...
```

静态变量示例

- 静态变量的存储
- 如何实现静态局部变量的一次初始化?

```
static int global_i = 2000;

void func(){
  static int local_i = 1000;
  printf("local static variable: %d\n", local_i);
  local_i++;
}
```

```
static int global_i = 2000;

void func(){
  static int local_i;
  local_i = 1000;
  printf("local static variable: %d\n", local_i);
  local_i++;
}
```

静态变量示例

- 静态变量的存储
- 如何实现静态局部变量的一次初始化?
  - 源代码对静态局部变量的初始化没有执行过

```
0804844d <func>:
 804844d:       55                      push   %ebp
 804844e:       89 e5                   mov    %esp,%ebp
 8048450:       83 ec 18                sub    $0x18,%esp
 8048453:       a1 28 a0 04 08          mov    0x804a028,%eax
 8048458:       89 44 24 04             mov    %eax,0x4(%esp)
 804845c:       c7 04 24 68 87 04 08    movl   $0x8048768,(%esp)
 8048463:       e8 a8 fe ff ff          call   8048310 <printf@plt>
 8048468:       a1 28 a0 04 08          mov    0x804a028,%eax
 804846d:       83 c0 01                add    $0x1,%eax
 8048470:       a3 28 a0 04 08          mov    %eax,0x804a028
 8048475:       c9                      leave
 8048476:       c3                      ret
```

作用域与存储空间

- 局部变量、静态局部变量、全局变量的区别
  - 存储空间上的区别（生命期）
    - 静态变量：目标文件、地址空间的数据段
    - 局部变量：地址空间的堆栈段（动态分配）
  - 作用域的区别
    - 编译器检查

缓冲区溢出

缓冲区溢出

- Memory Corruption

- 缓冲区溢出（Buffer overflow）
  - 维基百科定义:指当某个数据超过了处理程序限制的范围时，程序出现的异常操作

缓冲区溢出

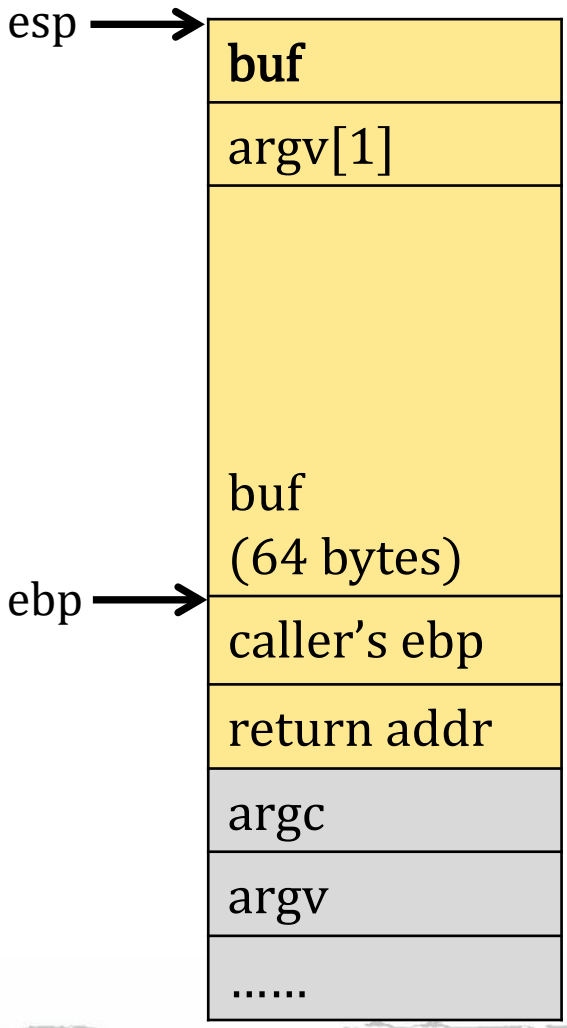
- 对于高级语言定义的每个对象，编译器在生成低级别的机器码时，会根据其类型及大小映射为内存中的多个对象
- 对变量的计算映射到对内存对象的操作
- 由于缺乏类型检查，内存对象之间可能出现互相破坏的问题

缓冲区溢出的漏洞利用

- 漏洞利用（Exploit）
  - An exploit is a piece of software, a chunk of data, or a sequence of commands that takes advantage of a vulnerability to cause unintended behavior to occur on computer software, hardware, or something electronic (usually computerized)
  - 利用这个漏洞而编写的攻击程序（program input）

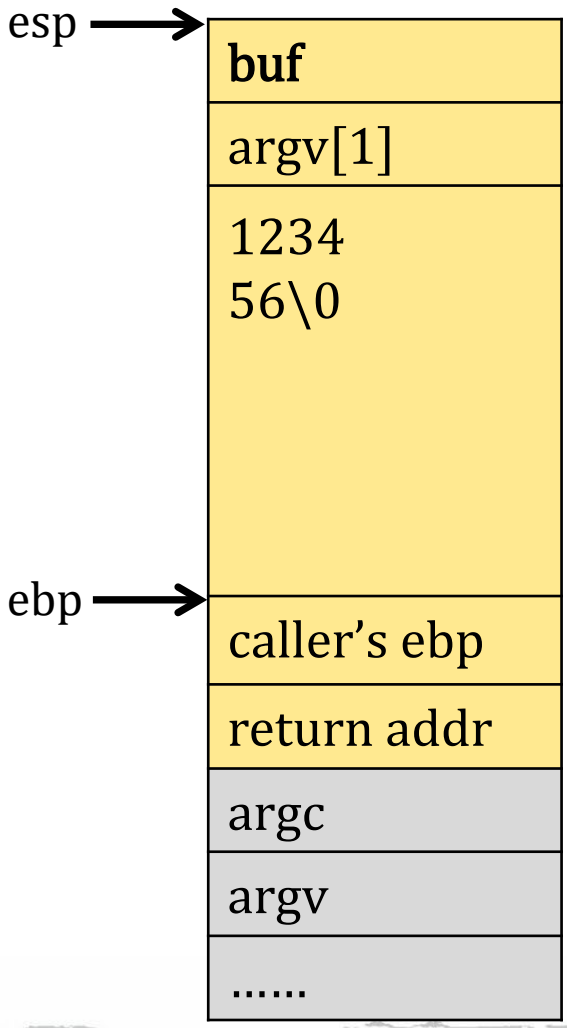
缓冲区溢出----例子

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



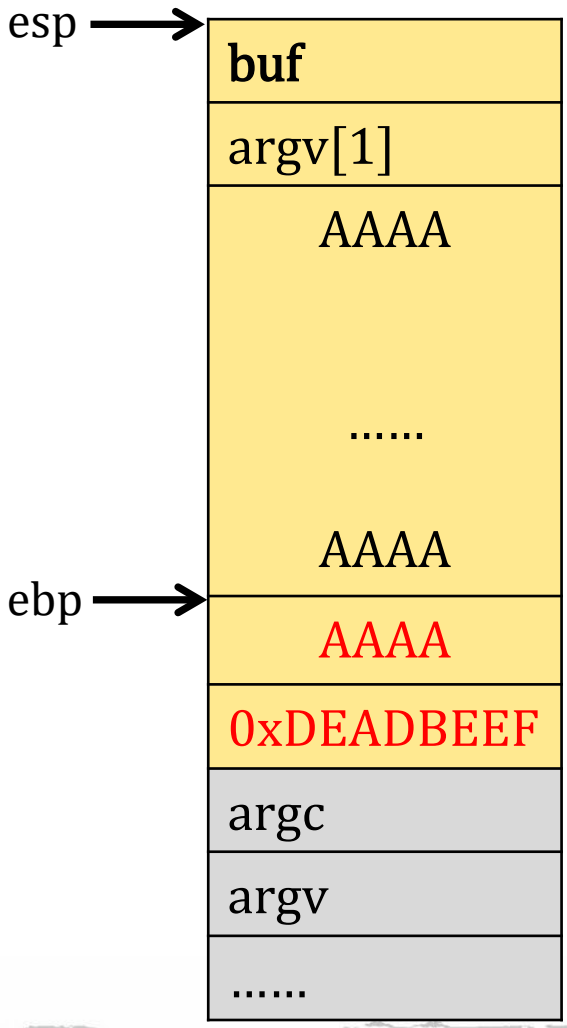
输入“123456”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```

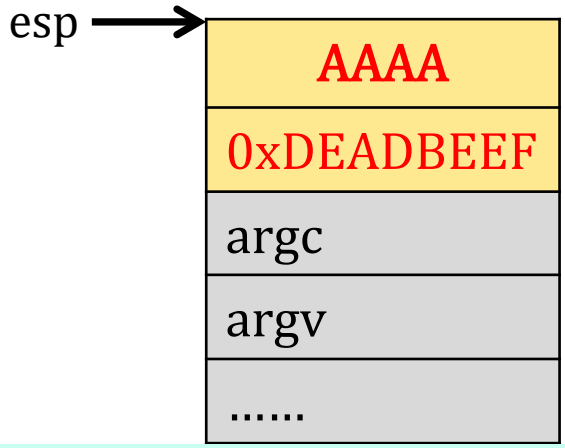


输入“A”x68. “\xEF\xBE\xAD\xDE”

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}

Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy@plt>
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



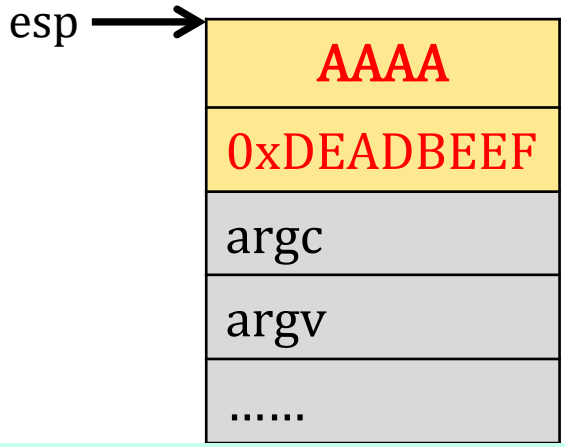
leave

1. mov %ebp, %esp
2. pop %ebp

ret

```
#include<string.h>
int main(int argc, char **argv) {
    char buf[64];
    strcpy(buf, argv[1]);
}
```

```
Dump of assembler code for function main:
   0x080483e4 <+0>:  push   %ebp
   0x080483e5 <+1>:  mov    %esp,%ebp
   0x080483e7 <+3>:  sub    $72,%esp
   0x080483ea <+6>:  mov    12(%ebp),%eax
   0x080483ed <+9>:  mov    4(%eax),%eax
   0x080483f0 <+12>: mov    %eax,4(%esp)
   0x080483f4 <+16>: lea    -64(%ebp),%eax
   0x080483f7 <+19>: mov    %eax,(%esp)
   0x080483fa <+22>: call   0x8048300 <strcpy
   0x080483ff <+27>: leave
   0x08048400 <+28>: ret
```



leave

1. mov %ebp, %esp
2. pop %ebp

ret

%ebp = AAAA

%eip = 0xDEADBEEF

缓冲区溢出的漏洞利用

- 最典型的
  - 利用构造数据填充栈
  - 栈上插入指令，如exec(“/bin/sh”)
  - 修改返回地址
  - 跳转到shellcode

从整形溢出到缓冲区溢出

- void * memcpy (void * destination, const void * source, size_t num);
  - Size_t: 无符号整形
  - 若出现有符号加法溢出，memcpy会将一个负数当做一个大的无符号整形，进而导致缓冲区溢出

格式化字符串

格式化字符串漏洞

- 格式化字符串函数
  - printf
  - fprintf
  - sprint
  - Syslog
  - vfprintf
  - syslog

printf(char *fmt, …)

指定了参数的类型及个数
(%d, %c, %x...)

可变参数

函数调用方式:cdecl, stdcall

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

格式化字符串漏洞

- 格式化字符串函数
  - printf
  - fprintf
  - sprint
  - Syslog
  - vfprintf
  - syslog

printf(char *fmt, …)

指定了参数的类型及个数
(%d, %c, %x...)

可变参数

对可变参数函数的调用，一定会遵循cdecl方式，为何?

可变参数函数

- 对于固定参数函数
  - 被调用者清楚被调用时的参数个数、类型
  - 从右往左压栈
  - 被调用者可利用栈帧寻参（ebp+4/8/16等）

- 对于可变参数的函数（格式化字符串）
  - 被调用者函数并不清楚参数的个数
  - 运行时利用栈帧动态的确定个数

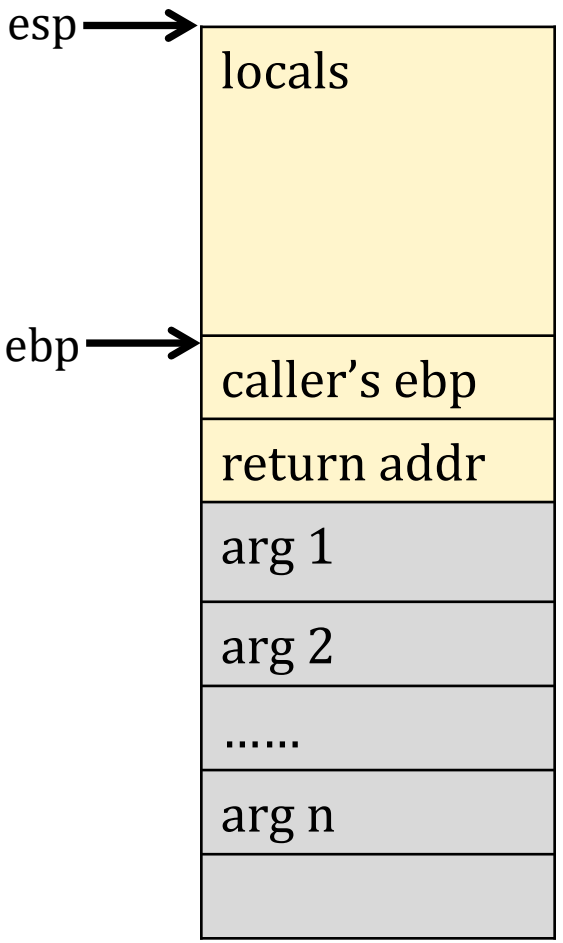
可变参数函数

- 设想：自己编码实现printf()
- 如何在函数体内寻参?

```
printf (“ hello world!\n”);
printf (“ %s hello world!\n”, buff);
printf (“%d %s hello world!\n”, &a, buff);
```

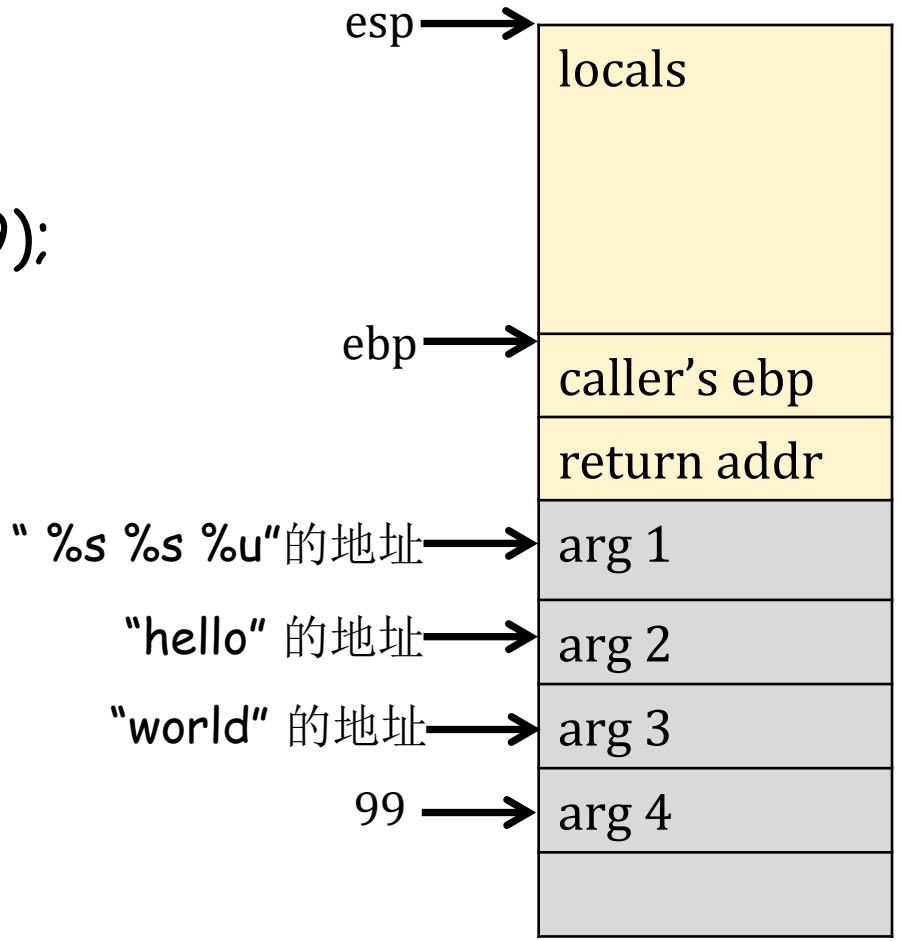
可变参数函数

- 格式化串制定参数的个数及类型
  - %d
  - %s
  - %x
- 通过栈基址，运行时动态的遍历参数



可变参数函数

```
Char s1[] = "hello";
Char s2[] = "world";
printf (" %s %s %u", s1, s2, 99);
```



格式化字符串漏洞

武漢大學
WUHAN UNIVERSITY

```
0    10    20    30    40    50    60
1  #include <stdio.h>
2
3  void main(int argc, char ** argv)
4  {
5      char buff[36] = {0};
6      int a = 0;
7      int b = 0;
8      FILE *fp;
9      char ch;
10
11     if( (fp = fopen(argv[1], "r")) != NULL )
12       fread(buff, 32, 1, fp);
13
14     printf("the addresses of a b buf are %x,%x,%x\n", &a, &b, buff);
15     a = 0;
16     printf(buff);
17     printf("a= %d\n", a);
18     if(fp) fclose(fp);
19 }
20
```

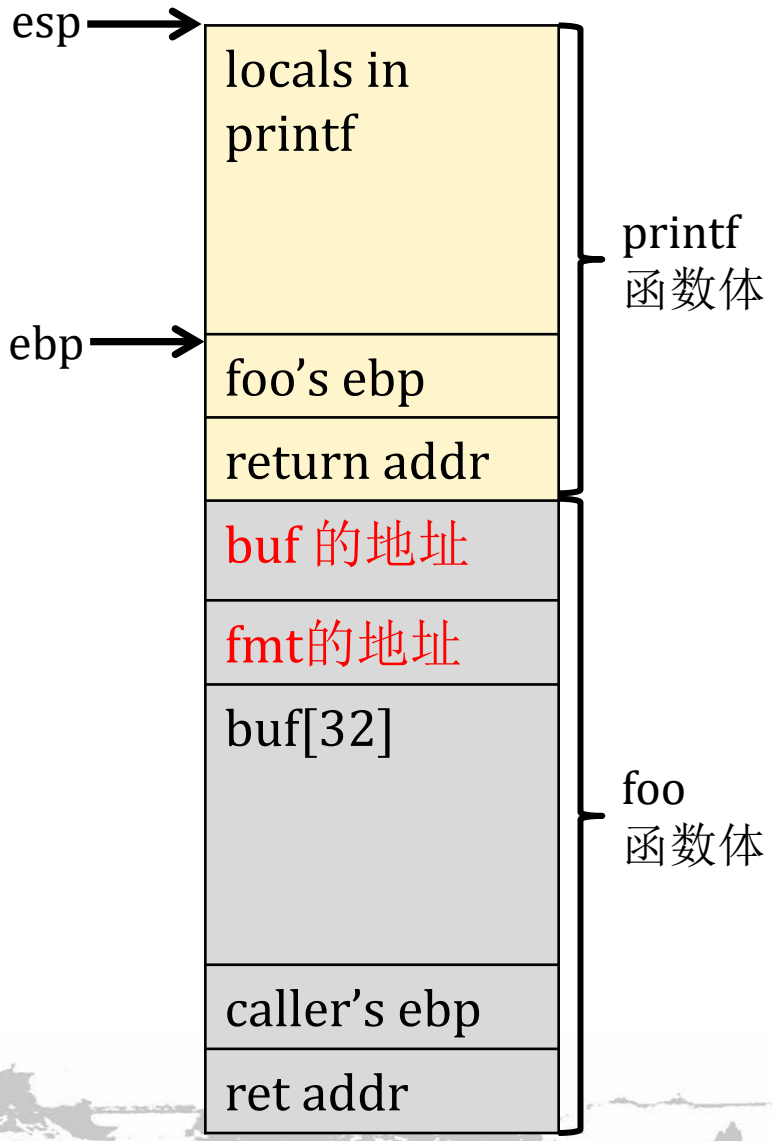
格式化字符串漏洞


```
int foo (char *fmt) {
        char buf[32];
        strcpy(buf, fmt);
        printf(buf);
 }
```

```
080483d4 <foo>:
 80483d4:       push   %ebp
 80483d5:       mov    %esp,%ebp
 80483d7:       sub    $0x28,%esp        ; allocate 40 bytes on stack
 80483da:       mov    0x8(%ebp),%eax    ; eax := M[ebp+8]  - addr of fmt
 80483dd:       mov    %eax,0x4(%esp)    ; M[esp+4] := eax  - push as arg 2
 80483e1:       lea    -0x20(%ebp),%eax  ; eax := ebp-32    - addr of buf
 80483e4:       mov    %eax,(%esp)       ; M[esp] := eax    - push as arg 1
 80483e7:       call   80482fc <strcpy@plt>
 80483ec:       lea    -0x20(%ebp),%eax  ; eax := ebp-32    - addr of buf again
 80483ef:       mov    %eax,(%esp)       ; M[esp] := eax    - push as arg 1
 80483f2:       call   804830c <printf@plt>
 80483f7:       leave
 80483f8:       ret
```

格式化字符串漏洞

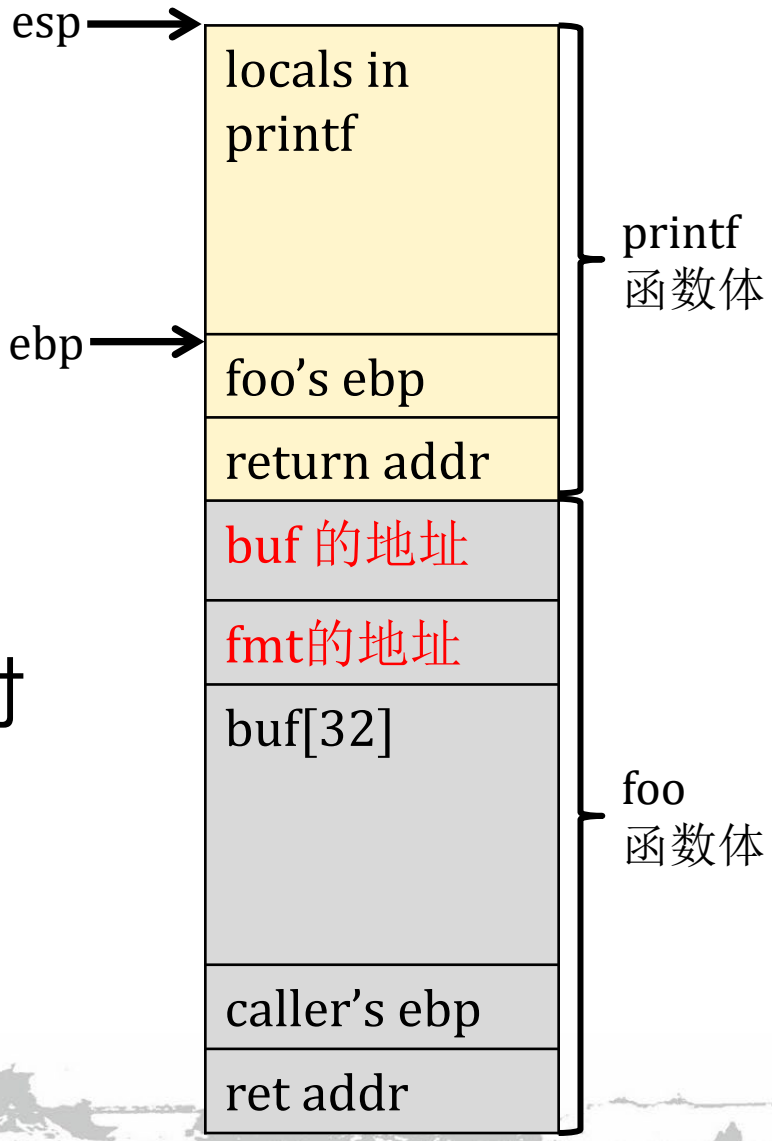
```
int foo (char *fmt) {
        char buf[32];
        strcpy(buf, fmt);
        printf(buf);
}
```



格式化字符串漏洞

```
int foo (char *fmt) {
        char buf[32];
        strcpy(buf, fmt);
        printf(buf);
 }
```

若fmt指向的字符串被蓄意构造如%x %x %x %x %x %x{8个}时会如何?

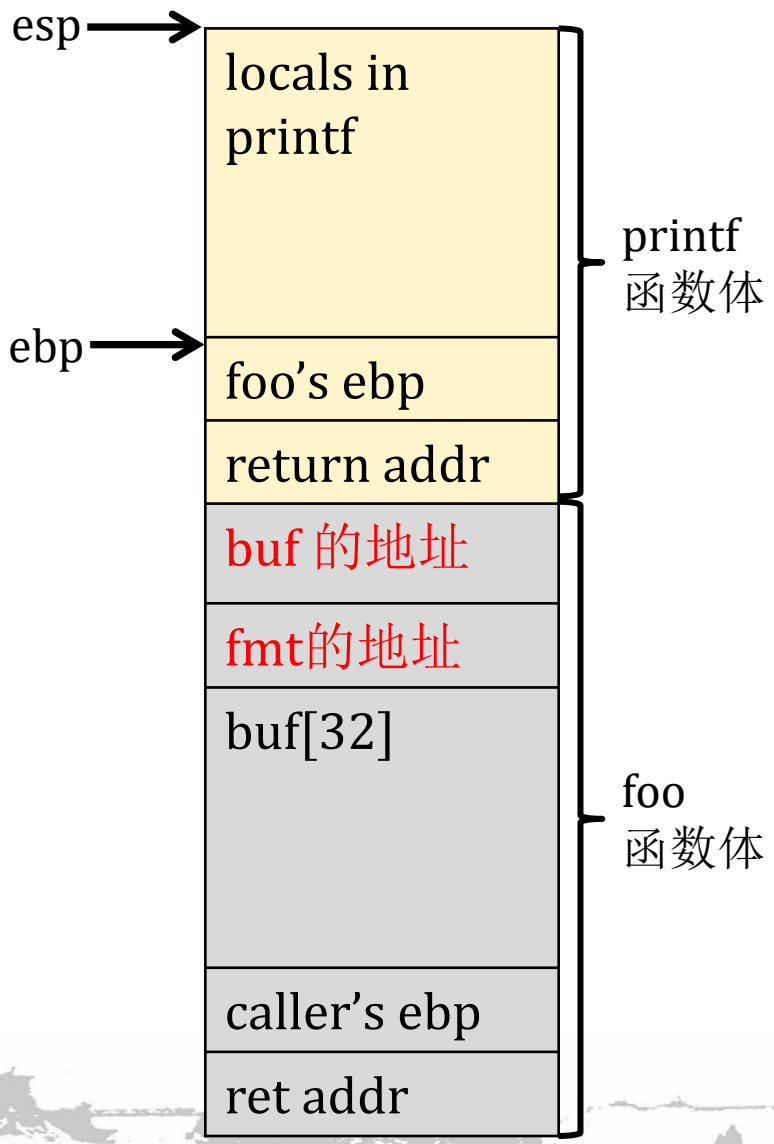


格式化字符串漏洞

```
int foo (char *fmt) {
        char buf[32];
        strcpy(buf, fmt);
        printf(buf);
 }
```

若fmt指向的字符串被蓄意构造

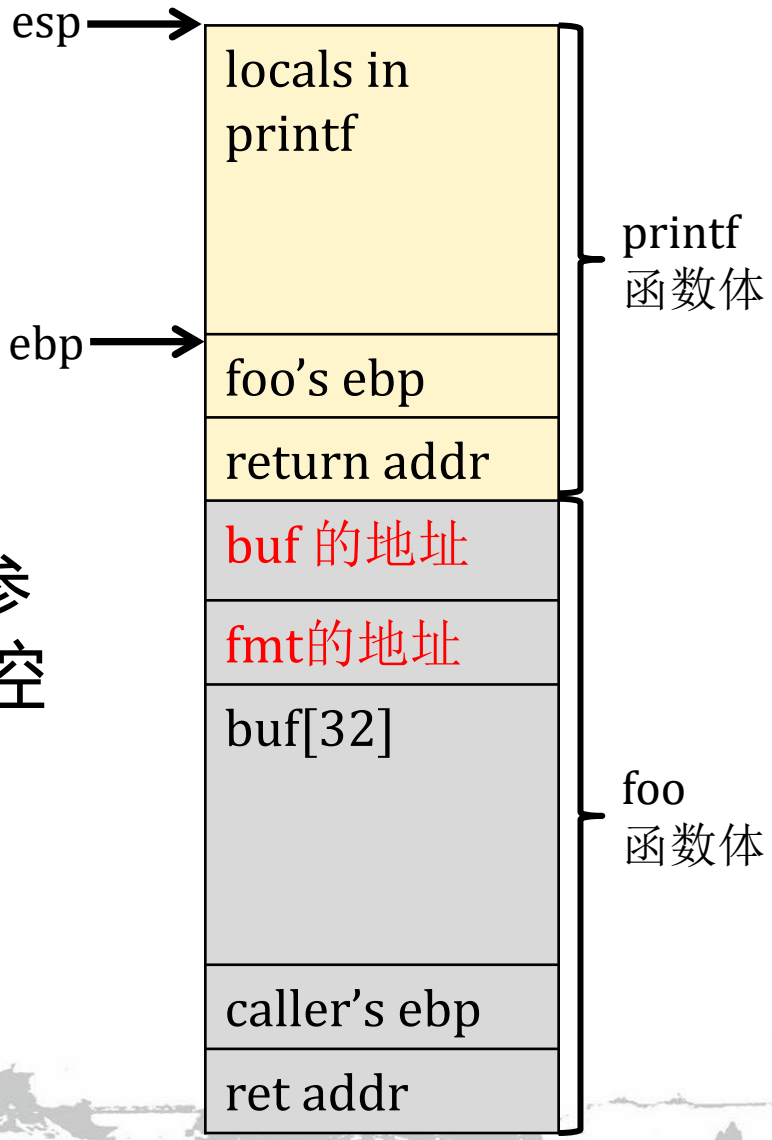
如指向 %x %s时又会如何?



格式化字符串漏洞

```
int foo (char *fmt) {
        char buf[32];
        strcpy(buf, fmt);
        printf(buf);
 }
```

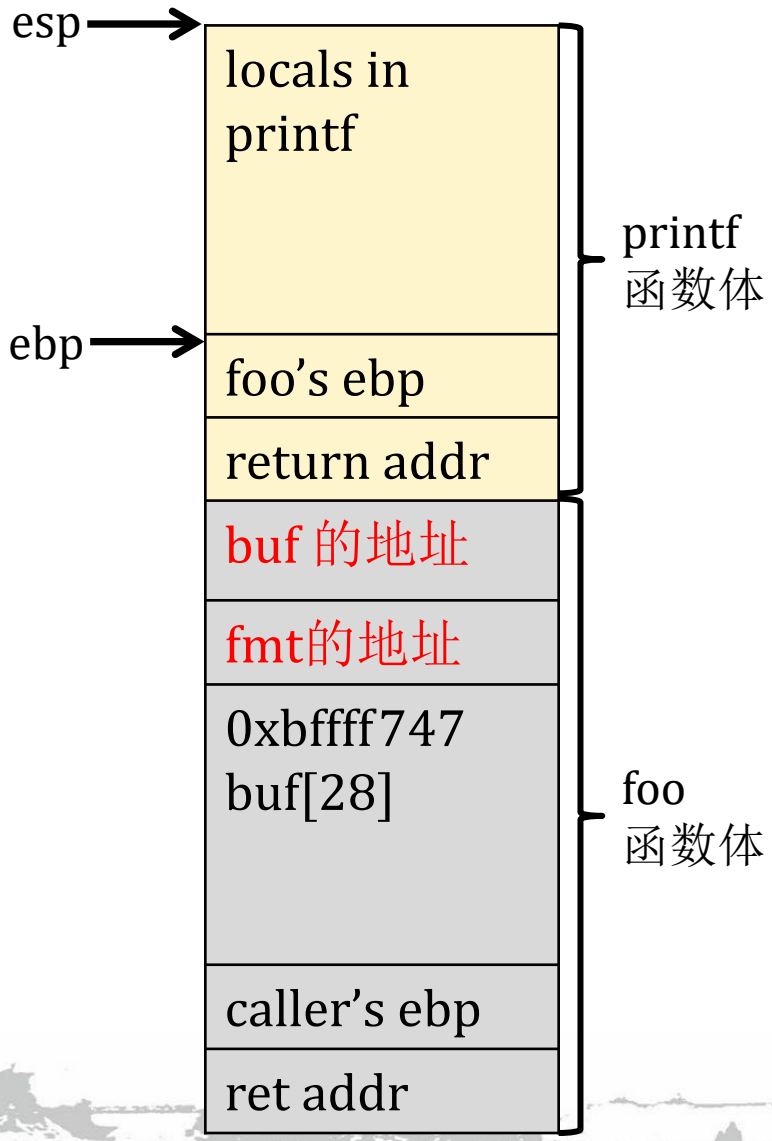
当前栈布局下buf 空间与printf参数相邻，因此，我们可以通过控制buf的数据来实现内存泄漏



格式化字符串漏洞

```
int foo (char *fmt) {
        char buf[32];
        strcpy(buf, fmt);
        printf(buf);
 }
```

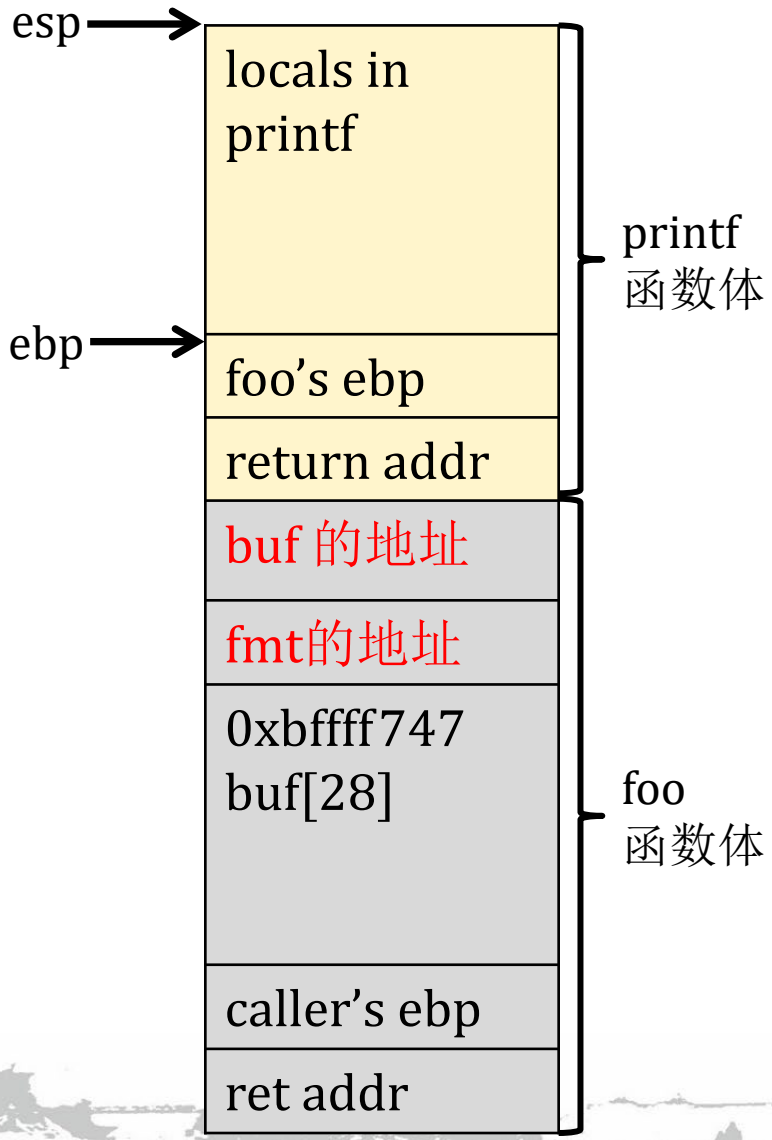
将fmt的内容填充为
\x47\xf7\xff\xbf%x%s,
则printf的输出是?



格式化字符串漏洞

```
int foo (char *fmt) {
        char buf[32];
        strcpy(buf, fmt);
        printf(buf);
 }
```

将fmt的内容填充为
\x47\xf7\xff\xbf %s,
则printf的输出又是?



格式化字符串漏洞的利用

- 内存泄露
- 任意地址写
  - %n
  - %{count}x
  - %{count}$n
- 利用sprintf等函数写缓冲区
  - Sprintf(buf, fmt, …);

内存泄漏

- 多样的格式化标记可以满足多种形式的泄露
  - 泄露栈
    - 变量值
    - ebp等可用来推测栈地址
    - 返回地址

printf 有意思的格式：%n

武漢大學
WUHAN UNIVERSITY

```
int a =0,b=0,c = 0;
printf(“%d,%d%n\n”,a,b,&c);
printf(“%d”,c);
```

打印的结果将是：

Line1： 0，0

Line2： 3

格式化字符串漏洞的利用

- 如何实现任意地址写任意值?
  - 地址如何构造?
    - 按格式化符号寻参，路漫漫其修远
  - 值如何构造?
    - 按字符个数计算值，愚公移山

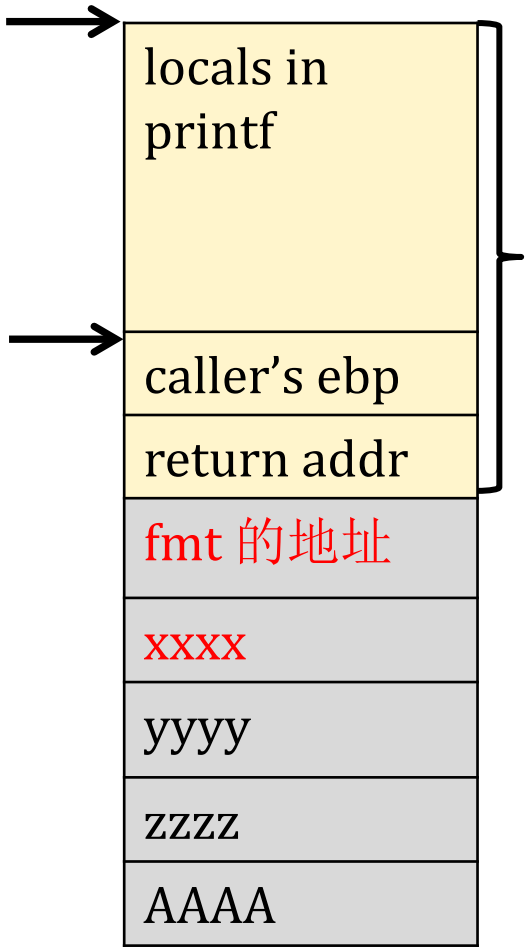
Printf里的$修饰符

武漢大學
WUHAN UNIVERSITY

- printf(“%p”)
  - 打印出栈中存放format下的第一个栈空间的16进制值
- printf(“%{x}$p”)
  - 打印出format下的第x个栈空间的16进制值
- %{x}$n
  - 把第x个栈空间存的值，当成是整形变量的地址，写入之前输出字符的长度

Printf里的$修饰符

- 如果我们的format是**%4$p**
  - 打印出0x41414141



Printf里的%{c}x修饰符

- %x
  - 打印出41414141
- %8x
  - 41414141
- Printf(“%05d”, 5, 42);
  - 00042

Printf里的$修饰符

武漢大學
WUHAN UNIVERSITY

- printf(“%{c}x”)
  - 构造C个字符，作为任意值

- %{x}$n
  - 定位第x个栈上的内存单元
  - 此处x是有符号数
  - 任意地址